NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DESIGN AND IMPLEMENTATION OF MODULE DRIVER AND OUTPUT ANALYZER GENERATOR

by

Gerald Anthony DePasquale

June 1990

Thesis Advisor: Valdis Berzins

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. REPORT SECURITY CLASSIFICATION: Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Naval Postgraduate School; 6b. OFFICE SYMBOL (If applicable): 37	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification): Design and Implementation of Module Driver and Output Analyzer Generator

12. PERSONAL AUTHOR(S): Gerald A. DePasquale

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM _____ TO _____	June 1990	281

16. SUPPLEMENTARY NOTATION: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number): Software Testing, Automatic Code Generation, Attribute Grammars, Software Reliability, Formal Specifications

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This thesis presents the Design and Implementation of a "Module Driver and Output Analyzer Generator" (MDOAG) for Spec functions. The generator translates specifications written in the Spec formal specification language into "Module Driver and Output Analyzers" (MDOA) written in Ada.

An MDOA is a testing tool which repeatedly calls the implementation of the function with input values provided by a generator program and reports instances for which the results fail to meet the specification. The classification of test results is carried out by Ada code that is automatically generated from the specification of the component to be tested.

The Kodiyak Application Generator, a fourth-generation attribute grammar tool, is used to implement the translation.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: X UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL: Dr. V. Berzins	22b. TELEPHONE (Include Area Code): (408) 646-2461; 22c. OFFICE SYMBOL: CS/Be (52Be)

DD Form 1473, JUN 86 — Previous editions are obsolete — SECURITY CLASSIFICATION OF THIS PAGE: Unclassified

Approved for public release; distribution is unlimited.

Design and Implementation of Module Driver and Output Analyzer Generator

by

Gerald Anthony DePasquale
Captain, United States Marine Corps
B.S., Purdue University, 1981

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1990

Author: _____
Gerald Anthony DePasquale

Approved by: _____
Valdis Berzins, Thesis Advisor

_____
Luqi, Second Reader

_____
Robert B. McGhee, Chairman,
Department of Computer Science

ABSTRACT

This thesis presents the Design and Implementation of a "Module Driver and Output Analyzer Generator" (MDOAG) for Spec functions. The generator translates specifications written in the Spec formal specification language into "Module Driver and Output Analyzers" (MDOA) written in Ada.

An MDOA is a testing tool which repeatedly calls the implementation of the function with input values provided by a generator program and reports instances for which the results fail to meet the specification. The classification of test results is carried out by Ada code that is automatically generated from the specification of the component to be tested.

The Kodiyak Application Generator, a fourth-generation attribute grammar tool, is used to implement the translation.

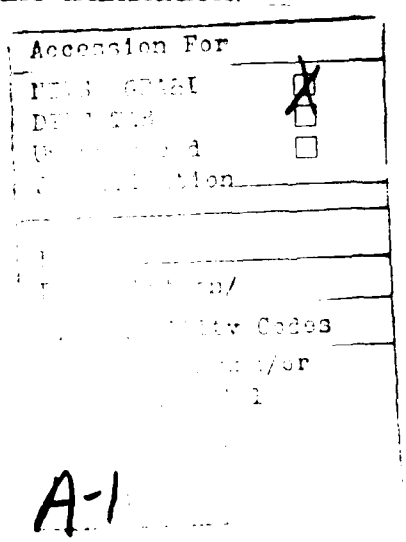

TABLE OF CONTENTS

LIST OF FIGURES

DEDICATION

To my Mom, Carolyn S. Anderson, the compassionate teacher who always has time to listen to her students' plights, who gives her students encouragement when no one else will, who infuses her students with the value of education, and who loves her students—My Mom and my teacher!

I. INTRODUCTION

A. MODULE DRIVER AND OUTPUT ANALYZER GENERATOR

A "Module Driver and Output Analyzer (MDOA)" is a system that repeatedly calls a module and reports the cases when the results of the call do not conform to the specification of the module. This research is concerned with the feasibility of implementing a "Module Driver and Output Analyzer Generator (MDOAG)" for a formal specification language using a fourth-generation attribute grammar tool.

The prototype MDOAG developed in this research automatically produces a MDOA from a specification. A schematic of the MDOAG is shown in Figure 1-1.

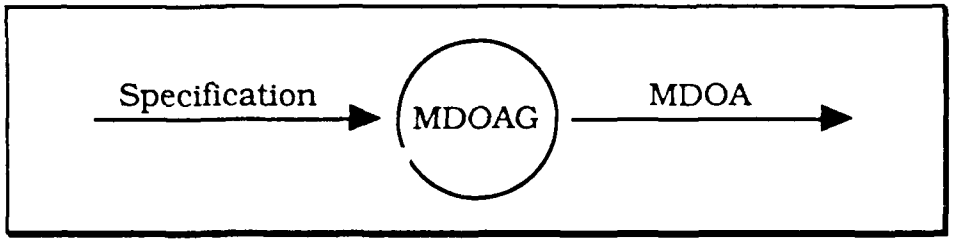

Figure 1-1. **Module Driver and Output Analyzer Generator**

The MDOAG "reads" a module specification and "writes" an MDOA for an arbitrary implementation of that module. The MDOA performs functional (black box) testing of the implementation. It provides an error report at the conclusion of the test. A schematic of the MDOA is provided in Figure 1-2.

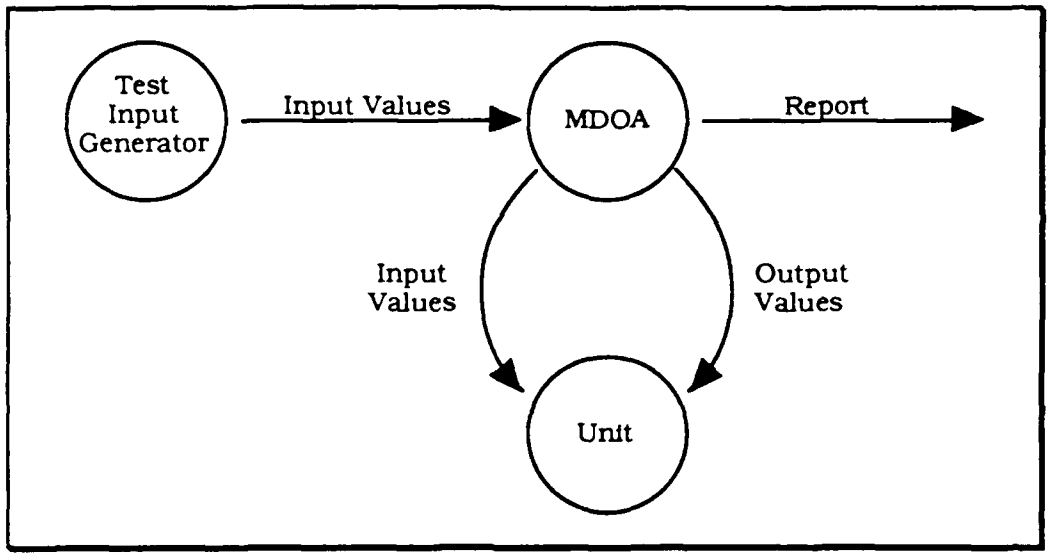

Figure 1-2. **Module Driver and Output Analyzer Generator**

In this system, the module specification is written in Spec [Ref. 1], a formal specification language described later. The MDOA is generated in Ada [Ref. 2]. The MDOA requires that the user, normally the test personnel, provide a test input generator. (*It does not require the corresponding* test output data that normally constitutes a test data set.) The module driver "calls" (executes) the implementation with each of the test input data generated and checks the output returned with the specified requirements. Every time an output value does not meet the specification, the test case (input and output values) and the portion of the specification violated are added to the report. At the conclusion of the test, the total number of tests conducted and errors found are also reported.

B. REQUIREMENTS FOR THE PROPOSED SYSTEM

There are numerous reasons a MDOAG is needed, only a few of which are provided in the following paragraphs.

1. To Reduce Time Spent in Unit Testing

The amount of time devoted to testing may be reduced using a MDOAG. The amount of time spent in testing varies depending on the type of system. In life critical-systems like those used in military systems, testing can consume as much as 80 percent of the total development effort [Ref. 3:p. 2]. An average estimate of time spent in quality assurance is 40 percent [Ref. 1:p. 4-62]. In such cases, there is a great potential for savings through automation of tasks currently performed manually.

2. To Reduce System Development Costs

The time savings gained through automation of the testing process should translate into significant savings in overall system costs. The U. S. Government spent $4 billion dollars in 1980 for major defense systems [Ref. 4]. Software costs are expected to reach $30 billion within the next few years [Ref. 4]. Researchers investigating ways to control these costs have suggested using automated aids to eliminate manual methods [Ref. 5].

3. To Encourage Unit Level Testing

An MDOAG encourages unit level testing because in automatically generates the code necessary to automatically conduct the unit tests. Systems should be tested incrementally, unit level first and gradually working towards system tests [Refs. 1, 3, 4, 6, 7]. However, due to delivery deadlines, developers sometimes try to save time by bypassing unit-level testing, moving directly to integration testing in hopes that both unit and integration bugs can be detected simultaneously. Often, this approach is unsuccessful. [Ref. 8:p. 115]

4. To Provide Unbiased Unit Tests

An MDOA, generated by an MDOAG, provides unbiased tests because the test code is generated automatically from the specification and the input data is randomly generated (an assumption). Separation of the implementation team and test team is encouraged as a means of reducing test bias. Often at the unit level, the two teams are the same people, which makes it very difficult to eliminate bias [Ref. 3:p. 6]. Regardless of who implements the unit and conducts test, the MDOA generated by an MDOAG will be free of bias and provide a sound indication of the unit's reliability.

5. To Increase the Reliability of Software

By automating the testing process, many more test cases may be run, significantly reducing the incidence of undetected errors.

C. SPEC SPECIFICATION LANGUAGE

1. General

Spec is a formal specification language used for describing the behavior of abstractions that detail a software system and its interactions [Ref. 8:p. 1]. It is intended primarily to represent black box specifications [Ref. 9:p. 75]. One of its many purposes is to provide the formalism (syntax and semantics) necessary to automate the testing process [Ref. 8: p. 1]. When applied properly, Spec yields modular, readable specifications.

2. Event Model

The event model is the semantic basis for Spec.

> The event model uses four primitives: modules, messages, events and alarms. A module is a black box that interacts with other modules by sending and receiving messages. A message is a data packet sent from one module to another. An event occurs instantaneously when a message is received by a module at a particular time. An alarm defines a time at a module and triggers temporal events at that module. [Ref. 9:p. 78]

a. Modules

Modules may be used to model software components (e.g., modules, units, subsystems, etc.). The behavior of a module is specified by describing its interface. An interface consists of a set of stimuli (events) it recognizes and their associated responses (sets of events). [Ref. 9:p. 78]

b. Messages

Messages may be used to model software component interactions (e.g., procedure calls, returns from procedures, Ada rendezvous, etc.). Messages have four attributes: origin (who sent the message), name (the name of the message), sequence of data values (the parameters), and condition (normal or exception). Figure 1-3 schematically depicts the interaction of two modules. $Module_1$ passes $message_1$ (stimulus) to $module_2$. $Module_2$ responds by passing $message_2$ (response) back to $module_1$. $Module_2$ knew to respond to $module_1$ by checking the origin attribute of $message_1$, which was $module_1$. This is important because a module may interact with several modules and must have a way of identifying which to respond to. [Ref. 9:p. 78]

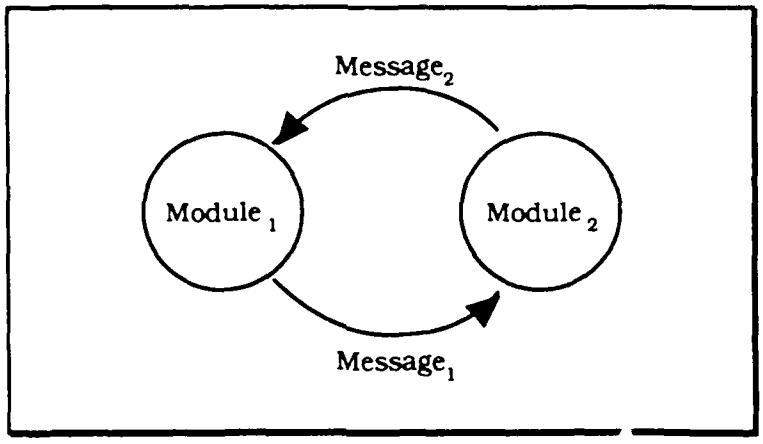

Figure 1-3. **Two Modules Communicating by Passing Messages**

Figure 1-3, as described above, could depict the interaction between a main procedure and a function. The main procedure ($module_1$) calls ($message_1$) the function ($module_2$) and the function returns results ($message_2$). Only the interface is depicted; there is no indication of the internal characteristics of the modules.

c. Events

An event records and describes the system's behavior. Events are identified by three properties: a module, a message, and a time. The time corresponds to the time a module received a message. [Ref. 9:p. 78]

An event can be reactive or temporal. A reactive event is an event which occurs in response to an external stimulus. A temporal event is one which occurs in response to an alarm set off by the module itself based on the absolute local time. [Ref. 9:p. 78]

d. Alarms

"Alarms represent discrete points in time when temporal events are triggered." [Ref. 9:p. 78] An alarm has a module, a message, and a time. An alarm causes the module to send itself a message at the designated time (modules have internal local clocks) [Ref. 9:p. 78]. Alarms illustrate a property of the event model– that all communications between modules must be explicit, even when a module communicates with itself. [Ref. 9:p. 78]

3. The Spec Language

This section is provided to give the reader some notion of the Spec language. Figure 1-4 is a specification for a "generic square root function" written in Spec [Ref. 9:p. 76]. It specifies proper behavior without providing implementation details. A narrative description of Figure 1-4 follows the discussion below.

```
FUNCTION square_root (precision:float SUCH THAT precision > 0.0)

    MESSAGE(x:float)
        WHEN x >= 0.0
            REPLY(y:float)
            WHERE y >= 0.0, approximates(y*y,x)
        OTHERWISE REPLY EXCEPTION imaginary_square_root

    CONCEPT approximates(r1 r2:real)
        --True if r1 is sufficiently accurate approximation of r2
        --The precision is relative rather than absolute
    VALUE(b:boolean)
    WHERE b <=> abs(r1-r2) <= abs(precision*r2)
END
```

Figure 1-4. **Specification for the Generic Square Root Function**

a. Spec Primitives

Spec has three primitive module types for specifying software components: functions, machines, and types [Ref. 9:p. 76]. Spec functions, machines, and types adhere to the conventional notions of "functions," "machines," and "types."

(1) Functions. A function has no memory (immutable), so a completely specified function calculates a single-valued mathematical function. Incompletely specified functions can be multiple-valued (non-deterministic).

(2) Machines. A machine maintains state information; consequently, a machine's response is dependent upon its computation history. [Ref. 9:p. 80]

(3) Types. A type defines an abstract data type. It provides the value set (data objects) of the type and a set of operations. The type manages the value set. Types can be immutable and mutable. An immutable type has a fixed value set and its operations cannot change the properties of the individual type instances. A mutable type can create and destroy type instances with internal states and can provide operations for changing them. [Ref. 9:p. 81–83]

b. Stimulus-Response

"The basic unit in a Spec module description specifies the required response to a stimulus." [Ref. 9:p. 76] Preconditions and post-conditions, written in predicate logic, specify the correct behavior of the module independently of its internal structure (i.e., Spec specifies correct behavior, not how to implement that behavior.).

c. Concepts

Spec concepts are used to decompose the predicate logic of a specification into understandable chunks in the same manner that subprograms are used in programs. Predicate logic can be difficult to read and understand if not digested in small pieces. Concepts make it possible to define predicates with meaningful names to enhance the readability of the specification. [Ref. 9:p. 77]

d. Sample Spec: Generic Square Root

As stated previously, Figure 1-4 shows a specification for a "generic square root." The Spec can be read as described in the following paragraphs. Selected Spec key words and delimiters (enclosed in brackets "[ ]") have been inserted into the narrative to indicate how they drive the interpretation.

(1) Module Header. [FUNCTION] "square_root" is a function, so it has no memory. It is generic [{ }] and has a single generic parameter "precision" of type "float," [SUCH THAT] the generic parameter "precision" is restricted to values that are greater than "0.0."

(2) Stimulus. Its stimulus [MESSAGE] is a single argument "x" of type "float" [( )].

(3) Response. The module must reply in one of two ways:

- [WHEN] the value of "x" is greater than or equal to "0.0", it must [REPLY] to the calling module with a single argument "y" of type "float [( )], [WHERE] the value "y" must be greater than or equal to "0.0" and its square (y*y) must "approximate" the input parameter "x."
- [OTHERWISE] (i.e., when the value of "x" is not greater than "0.0"), it must [REPLY] with the [EXCEPTION] "imaginary_square_root."

(4) Concepts. The [CONCEPT] "approximates" defines the predicate "approximates" used above. Two values "r1" and "r2" of type float "approximate" each other when the absolute value of their difference is less than or equal to the absolute value of "r2" multiplied by the value of the parameter "precision" [WHERE]. (The "approximates" concept may be thought of as a function returning a boolean value.)

D. ATTRIBUTE GRAMMARS AND THE KODIYAK APPLICATION GENERATOR

Attribute grammars are the theoretical foundation for this research. The Kodiyak Application Generator is a tool that implements that foundation. In this research, an attribute grammar (based on the Spec specification language) and the Kodiyak Application Generator are used to define and automate the process of "translating" module specifications into MDOAs.

1. Attribute Grammars

Knuth introduced attribute grammars as a means of defining the formal semantics of context free languages. Intuitively, all legal strings of a grammar can be represented by parse trees, where the root of the parse tree is the start symbol; the interior nodes are the non-terminals; and the leaf nodes are the terminal symbols. Knuth showed that by (1) assigning attributes to the nodes of the tree and (2) establishing rules (semantic functions) by which those attributes derive their meaning, semantic information about the parsed string could be passed about the

tree.[1,2] Collectively, that information conveys the semantics of the string [Ref. 10]. In practice, attributes are defined for the terminals and non-terminals of the grammar, and semantic functions are associated with the production rules of the grammar. For any given string, the parse tree is constructed, and all attributes associated with that parse tree are evaluated. When attribute evaluation is complete, "the defined attributes at the root of the tree constitute the 'meaning' corresponding to the derivation tree." [Ref. 10]

2. Kodiyak Application Generator

a. General

The Kodiyak Application Generator is a language for constructing translators modeled after Knuth's description of attributed grammars. It has facilities for describing a lexical scanner, an LALR(1) grammar, and attribute definition equations. [Ref. 11]

The description that follows provides only that information required to understand the code in this work. For further information consult Reference 11.

b. Comments

An exclamation point ("!") introduces a comment which runs to end-of-line.

---

[1]Attributes that derive their meaning solely from descendant nodes are known as synthesized attributes [Ref. 10].

[2]Attributes that derive their meaning solely from ancestor nodes are known as inherited attributes [Ref. 10].

c. Program Format

AG programs consist of three sections. The program layout and the purposes of the three sections are shown in Figure 1-5.

```
! SECTION 1: LEXICAL SCANNER
  ! PURPOSES:
      ! specify the terminal symbols of the language.
      ! specify rules for translating source text into
      ! terminal symbols.
      ! establish operator precedences.
%% ! SECTION DELIMITER
! SECTION 2: ATTRIBUTE DECLARATION SECTION
  ! PURPOSES:
      ! Name the attributes.
      ! Define their types.
%% ! SECTION DELIMITER
! SECTION 3: GRAMMAR AND ATTRIBUTE EQUATIONS
  ! PURPOSES:
      ! Define the grammar.
      ! Define the attribute equations which describe the
      ! semantics of the translation.
```

Figure 1-5. **AG Program Format and Section Purposes**

d. Lexical Scanner Section

> The lexical scanner...defines a set of substitutions to be performed on the input text. Named terminal symbols are associated with regular expressions. Input is scanned for text which matches these regular expressions. If such a match is found, the text is deleted and replaced with an occurrence of the associated terminal symbol. [Ref. 11:p. 3]

AG accepts regular expressions as recognized by Lex [Ref. 11:p. 4; Ref. 12]. AG also provides token definitions to enhance readability [Ref. 11:p. 4]. Code fragments from the lexical section of Appendix A are provided in Figure 1-6. The numbers enclosed in asterisks (e.g., *1*) have been added to assist in explaining the code. They do not appear in the actual code. Statement *1* defines the token "Digit" to be a single ASCII character between "0" and "9". Statement *2* defines "Int" (integer) to be one or more of the tokens Digit. The curly braces ("{ }") indicate that a previously defined token is being used in the current definition. Statement *3* defines the token "AND" to be the ampersand ("&"). Statement *4* defines a REAL_LITERAL to be the string composed of an Int token followed by a decimal point (".") and another Int token. Operator precedences and associativities for AND, "+," and "-" are established by statements *5* and *6*. Precedence is established by the line on which the operators appear. Operators on the same line have the same precedence. The operators on different lines derive their precedences from their line number. Lower line number implies lower precedence. Hence, the operator "AND" has a lower precedence than "+" and "-." The operators "+" and "-" have the same precedence. "%left" implies left associativity for the line. Hence the expression "2 + 3 + 5" is the same as "(2 + 3) + 5."

e. Attribute Declaration Section

"The attribute declaration section consists of attribute declarations for all non-terminals and named terminals." [Ref.11:p. 7]

```
! LEXICAL SECTION: SOME EXAMPLES
! definitions of lexical classes

%define Digit          :[0-9]                    *1*
%define Int            :{Digit}+                 *2*

! definitions of compound symbols and keywords

AND                    :"&"                      *3*
REAL_LITERAL           :{Int} "." {Int}          *4*

! operator precedences

%left                  AND;                      *5*
%left                  '+', '-';                 *6*
```

Figure 1-6. **Code Fragments From Lexical Section of Appendix A**

Kodiyak has two primitive data types: string and int (integer) [Ref. 11:p. 7]. They have the conventional interpretation. Strings may contain control characters (e.g., newlines, tabs, etc.). They are introduced in the string with a backslash, like in the "C" programming language.

A "map" is the only higher order type in Kodiyak. A map "maps" one primitive type onto another. A map is like a lookup function with an entry operator and a lookup operator. Given a "key," it returns the value pointed to by that key. For instance, string -> int is a map from

strings to integers. Given a particular string, the map returns a integer associated with that string.

AG has two special attributes: %text and %line. They provide the actual text of the token as scanned and the line number of the source on which it appeared.

Some samples of attribute declarations are provided in Figure 1-7. The non-terminal "start" has two attributes: "main" and "lines_of_code," statements *1* and *2*, respectively. Main has type string and lines_of_code has type integer. The "message" non-terminal has one attribute named "table." It is a map of strings to integers, statement *3*. The terminal symbol "REAL_LITERAL" has the special attribute %text. It will be set to the string representing the actual REAL_LITERAL scanned (e.g., "1.0", "2.50", etc.).

```
! ATTRIBUTE DECLARATION SECTION: SOME EXAMPLES

%%
start {
     main:string;              *1*
     lines_of_code:int;        *2*
};

message{
     table:string->int;        *3*
};

REAL_LITERAL{
     %text:string;             *4*
};
%%
```

Figure 1-7. **Sample Attribute Declarations**

f. Attribute Grammar Section

"The attribute grammar section defines the syntax and semantics of the translation." [Ref. 11:p.9] The grammar is defined in a notation similar to BNF augmented with the semantics of the translation. For each production alternative of a production rule in the grammar, semantic functions are defined and enclosed between curly braces ("{ }"). Figure 1-8 provides an example of a production rule augmented with semantic functions. The "messages" production rule has two possible productions: (1) the "messages" non-terminal followed by the "message" non-terminal or (2) the null production. Semantic function *1* is the semantic function for the first alternative and will be evaluated if the parse generates the first alternative. Semantic functions *2* and *3* are the semantic functions for the second alternative and will be evaluated if the parse generates null.

```
messages
    : messages message
        {   messages[1].parm_specs=message.parm_specs;          *1*
        }
    |
        {   messages.parm_specs="";                             *2*
            messages[1].r_parm=                                 *3*
                messages[2].parm_specs == ""
                -> ""
                # messages[2].r_parm;
        }
```

Figure 1-8. **Production Rule Augmented with Semantic Functions**

Semantic function *1* states that the attribute "parm_specs" of the first occurrence of the non-terminal "messages" (the instance in the left-hand side of the production, denoted by messages[1].parm_specs) is equal to the value of the "parm_specs" attribute of the "message" non-terminal (denoted by message.parm_specs). References to the "parm_spec" attribute of the second "messages" non-terminal would be denoted by messages[2].parm_specs (i.e., the second occurrence of). When the brackets ("[ ]") are omitted, the first occurrence of the non-terminal is assumed.

Semantic function *2* states that the "parm_specs" attribute of the first occurrence of the "messages" non-terminal evaluates to the empty string (" ").

Semantic function *3* illustrates the conditional (if-then-else) construct. The example can be interpreted as shown in Figure 1-9. The first attribute in the AG construct is the one to which an assignment will be made. The next line in the construct is a conditional expression. If the condition is true, the first attribute is assigned the value after the "->" (then) symbol. If the condition is false, the first attribute is assigned the value after the "#" (else) symbol. [Ref. 11:p. 14]

Semantic functions *1*, *2*, and *3* are examples of information being passed up the parse tree (synthesized) from a sibling node to a parent node. In the same manner, information could be passed down the tree (inherited) by changing the order of the non_terminals. (e.g., message.parm_spec = messages[1].parm_spec would pass the value of

```
if (messages[2].r_parm == "") then
    (messages[1].r_parm = "")
else
    (messages[1].r_parm = messages[2].r_parm)
end if;
```

Figure 1-9. **Interpretation of AG if-then-else Construct of Figure 1-8**

the "parm_spec" attribute down the parse tree). Information can also be passed from sibling to sibling by simply defining a semantic function passing information between attributes on the right-hand side of the production rule (e.g., messages[2].parm_specs = message.parm_spec would pass information to the right in the messages[1] subtree.). AG provides the operators listed in Figure 1-10. Most of the operators carry their usual meaning. However, the map join operator requires some explanation.

> The join operation takes two maps of the same type...and constructs a new map. The new map is defined everywhere either of the other maps is, and is undefined wherever both of the other maps are. Every pair in the first map is included in the resulting map. Every pair in the second map that does not have a key that occurs as a key in the first map also appears in the resulting map. [Ref. 11:p.15]

Two standard functions provided by AG were used in the MDOAG code: %outfile(file_name:string, val:string), which writes the string val to the file named file_name, and %errfile(file_name:string, val:string), which does the same thing except "file_name" is the name of

Arithmetic Operators

Operator	Meaning
"+"	addition
"-"	subtraction
"*"	multiplication
"/"	division
"%"	modulus
"-"	negation

String Operators

"[ ]"	concatenate enclosed strings

Map Operators

"+|"	join: joins two maps of the same type
"non_term.map_name(key)"	apply: given a non_terminal "non_term" with a map attribute "map_name" and a "key," returns value associated with the "key"

Relational Operators

"<"	less than
">"	greater than
"=="	equal
"<>"	unequal
"<="	less than or equal
">="	greater than or equal

Logical Operators

"&&"	and
"||"	or
"~"	negation

Figure 1-10. **AG Operators**

the error file [Ref. 11:p. 19]. The first rule in the section is for the start symbol. The attributes of the start symbol will yield the semantic meaning of the translation. All output is generated by the output functions defined for the start symbol. [Ref. 11:p. 24]

II. PREVIOUS WORK

Presented below is a limited sampling of work that most closely approximates the work done in this research. The first section discusses two "module drivers" and the second mentions two tools that have been implemented using attribute grammar tools.

A. MODULE DRIVERS

Research, development, and implementation of automatic module driver tools was conducted as early as 1974 [Ref. 13]. Two representative tools are described below. Note that both of those tools require test personnel to provide a complete test case (module inputs and expected outputs). Further, they are not generated automatically from the specification; consequently, the results are valid only if the tester's interpretation of the requirements is correct.

1. Automatic Unit Test (AUT) Program

The Automated Unit Test Program, released by IBM in 1975, is one of the first automatic software test drivers developed. It conducts black box testing and automatically tests the "object module" of the implementation (i.e., it is independent of the source code the module is written in) [Ref. 14]. A schematic of the system is given in Figure 2-1.

To conduct a test, a test procedure must be written in Module Interface Language-Specific (MIL-S), a test language also developed by IBM. The test procedure provides a sequence of test cases to be executed

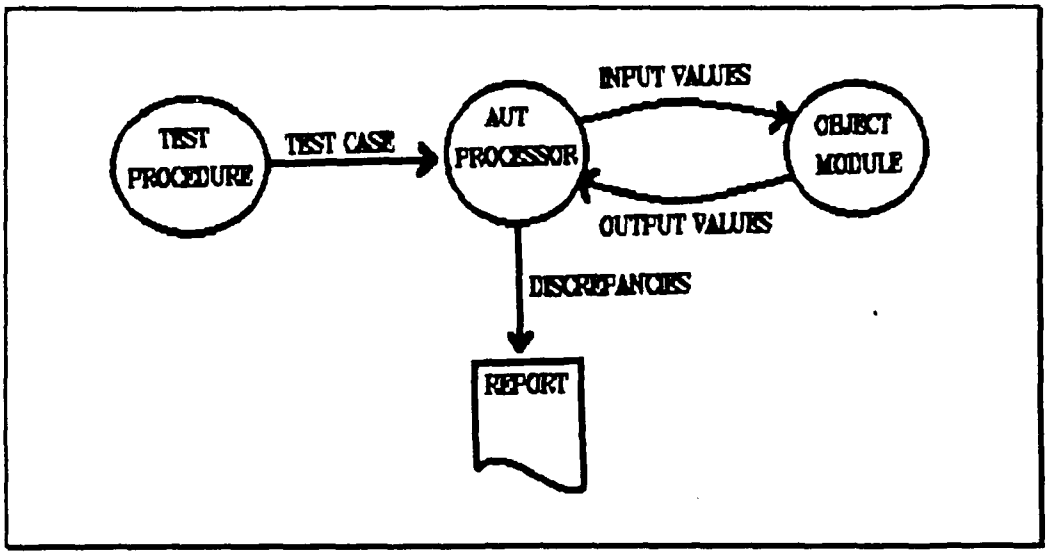

Figure 2-1. **Schematic of IBM's Automatic Unit Test Program**

by the AUT processor. Each test case includes test input data and its corresponding output data. For each test case in the test procedure, the AUT processor calls the object module with the input values. The object module then computes and returns its results to the AUT processor. Then the AUT processor checks the results against the correct output data of test case and records any discrepancies noted. [Ref. 14]

The AUT is criticized because the test language, MIL-S, is a low-level assembler-like language and it has no facilities for modeling I/O devices and files. [Ref. 14]

2. Fortran Test Procedure Language (TPL/F)

TPL/F is an automated test system developed by General Electric. It operates on the implementation's source code (Fortran). The language allows test cases to be specified in terms of the implementation's internal structure (i.e., it is capable of conducting clear box testing).

Further, the system gathers statistics of testing thoroughness such as percentage of statements executed and branches traversed. [Ref. 14]

The general testing process is similar to the AUT. Like the AUT, this system also requires that a test procedure be written which includes input and output data, but it also allows the procedure to specifically conduct tests over segments of the implementation source code if desired. To support that feature, the system has what it calls a target-program translator. Basically, it is a source code parser which enables test code insertion. Like the AUT, test cases are conducted and the results are written. Unlike the AUT, the test results include test coverage statistics. [Ref. 14]

B. TOOLS IMPLEMENTED WITH ATTRIBUTE GRAMMAR TOOLS

Attribute grammar tools have been used successfully to implement several systems. Two examples are provided below.

1. Language Translator for the Computer-Aided Prototyping System (CAPS)

The Language Translator for CAPS is written using the Kodiyak Application Generator. It translates prototype specifications written in the Prototype System Description Language (PSDL) into a set of Ada procedures and packages. Altizer, the author of the translator, presents a "template translation methodology" that is used to produce the Module Driver and Output Analyzer Generator of this research [Ref. 15]. The template methodology is explained in Chapter IV and is used in the research reported here.

2. Specification Language Type Checker

A specification language type checker is also implemented using the Kodiyak Application Generator. The system performs name analysis and error reporting for specifications written in Spec [Ref. 16].

The Module Driver and Output Analyzer Generator of this research assumes there are no type errors in the input Specs. In effect, the result of this research is an extension of the total Spec environment, which includes the type checker.

III. DESIGN OF THE MDOAG

A. SUBSET OF THE SPEC LANGUAGE IMPLEMENTED

The subset of the Spec language treated by this thesis corresponds roughly to Spec functions that can be implemented as Ada functions or procedures. This set consists of single service functions which receive a single anonymous message and reply with one or more parameters or an exception. The square root function of Figure 1-4 is a typical example.

A more precise description of the subset is provided in Appendix B, which contains the set of Spec production rules that have been implemented by this system. In addition, the user's manual (Appendix C) describes the minimum details necessary to determine whether a specification may be tested using the system.

B. DESIGN OF THE RUN-TIME SYSTEM

Figure 1-1 shows that the Module Driver and Output Analyzer Generator (MDOAG) reads in a Spec and generates a Module Driver and Output Analyzer (MDOA) for that Spec. However, it actually generates an incomplete MDOA. It generates eight files containing Ada source code for all but two of the components of the MDOA. The user must supply one of the remaining components representing the code to be tested, and the other is resident in the environment. Further, the user is required to complete at least one component (possibly more) prior to compiling the MDOA into executable code. The executable MDOA reads in a file containing test parameters (supplied by the user) and outputs the results of

the test. Refer to the user's manual (Appendix C) for details concerning the user interface.

The components of the MDOA are presented in the dependency diagram of Figure 3-1. All the components except MDOAG_LIB and IMPLEMENTATION are generated by the MDOAG. MDOAG_LIB is a standard library component. IMPLEMENTATION is the component to be tested. ITERATORS are generated as required, depending on the Spec.

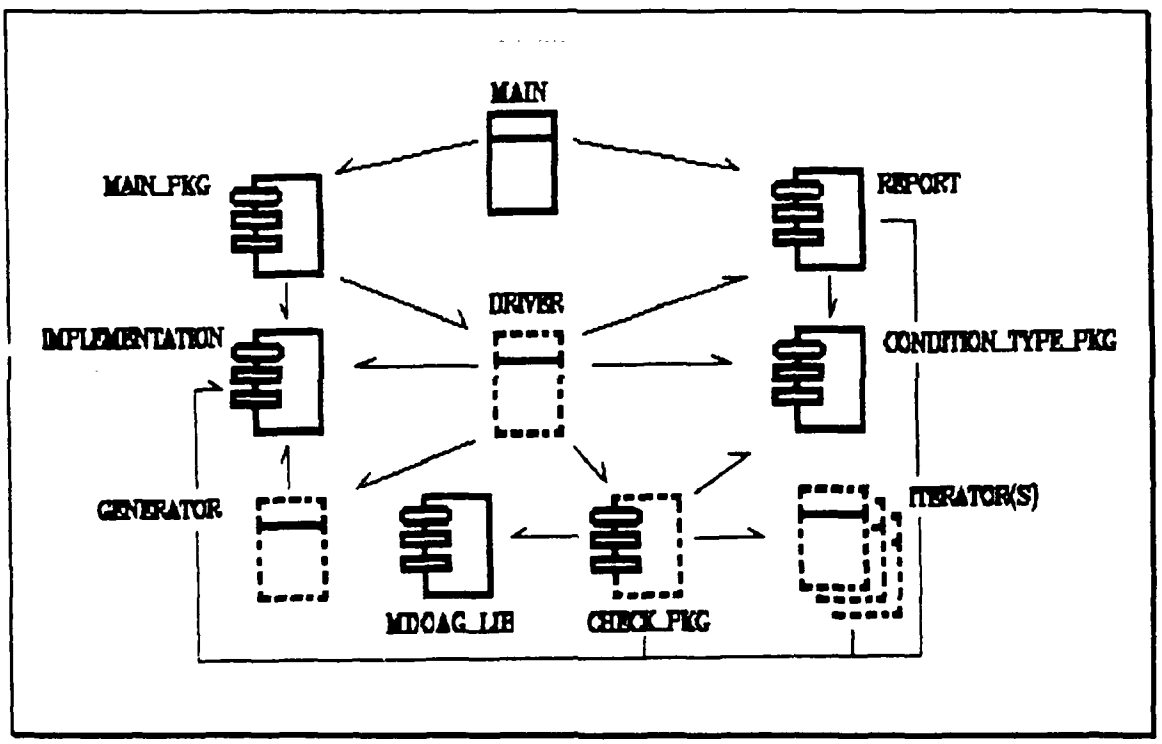

Figure 3-1. **Architecture of the Module Driver and Output Analyzer**

The GENERATOR and ITERATORS must be completed by the user. For these components, the MDOAG generates a shell providing the proper interface. The user must add the code necessary to generate the

input data sets and ranges for any quantifiers appearing in the specifications.

Examples of each component produced by the system are contained in Appendix D. When referring to the components, be aware that the DRIVER, CHECK_PKG and ITERATOR components generated by the MDOAG contain "m4 macros" which require expansion via the UNIX m4 macro processor prior to Ada compilation; consequently, two files are given for the same component (e.g., DRIVER.M4 and DRIVER correspond to the DRIVER before macro expansion and after expansion, respectively.). The macros are more readable than the expanded Ada code (see DRIVER and GENERATOR sections for more detail). Descriptions of each component are provided in the next several sections.

Figure 3-2 is a detailed data flow diagram of the MDOA. It is provided to supplement the following discussions and as a "master" diagram of the MDOA. Subprogram calls are indicated in upper case and are located above and to the left of the arrows. Data components are indicated in lower case and are located below and to the right of the arrows. A single arrow was used to represent two distinct MAIN-REPORT interactions and three distinct DRIVER-REPORT interactions to save space (e.g., OPEN, CLOSE). No specific CHECK_PKG-MDOAG_LIB interactions are indicated. The dashed arrow represents an exception message.

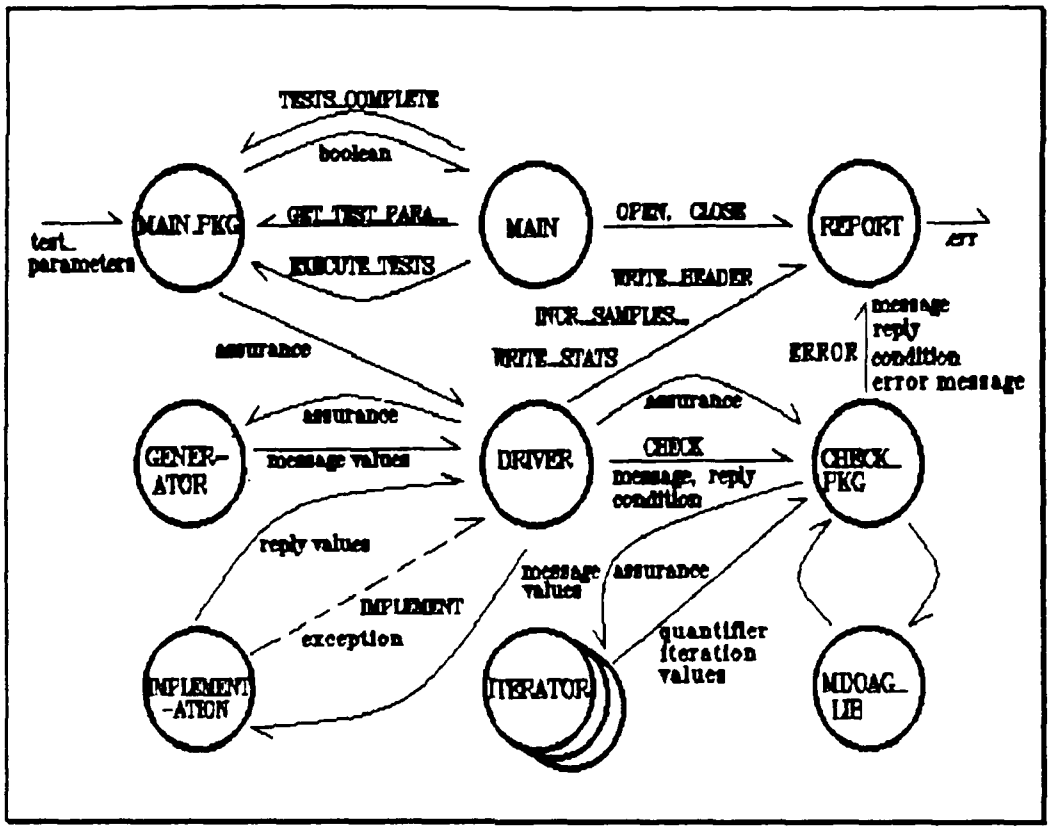

Figure 3-2. **Module Driver and Output Analyzer Data Flow Diagram**

1. MAIN

MAIN is the "main" procedure of the system. MAIN's architecture is provided in Figure 3-3, which graphically illustrates its dependencies and their precise nature. MAIN depends upon REPORT and MAIN_PKG for the services they render, as indicated (e.g., REPORT provides OPEN). The general functionality of MAIN can be determined by reading the services from left to right. The services whose symbols are filled (e.g., GET_TEST_PARAMETERS) are contained inside a loop which is controlled by the service whose symbol contains the dot (i.e., TESTS_COMPLETE).

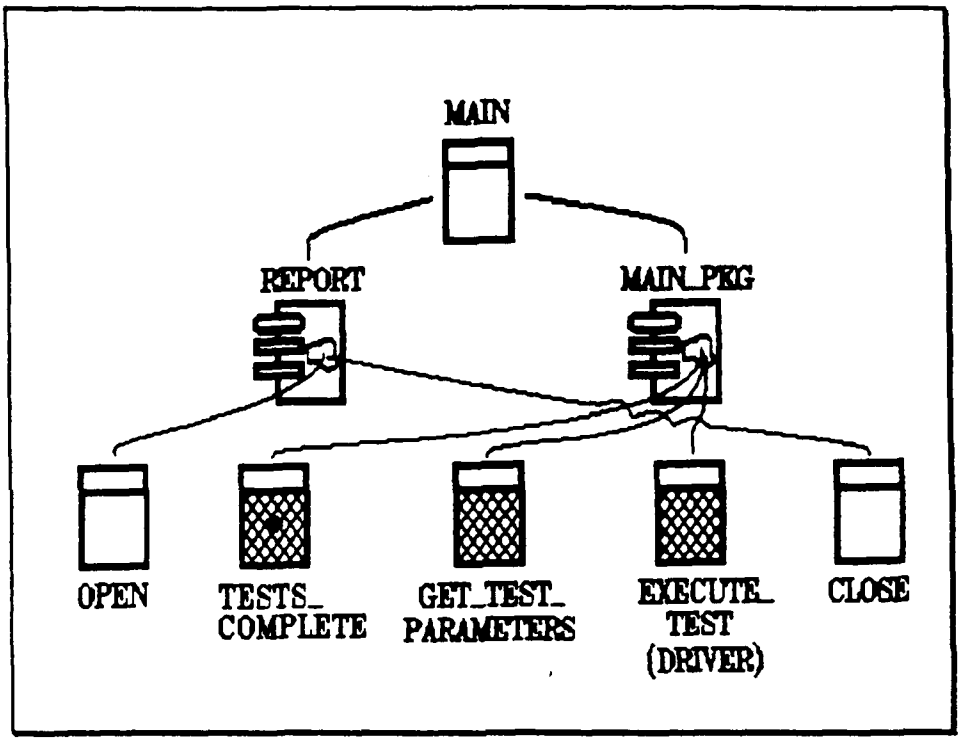

Figure 3-3. **MAIN Architecture**

Procedure MAIN opens the error report. Then it repeatedly gets test parameters and executes the test, until the tests are complete. Executing a test adds information to the error report if the results of the test do not conform to the specification. Then it closes the error report.

The top row of modules in Figure 3-2 shows the communication of procedure MAIN.

2. MAIN_PKG

MAIN_PKG hides some of the implementation details of procedure MAIN. It has three subprocedures: TESTS_COMPLETE, GET_TEST_PARAMETERS and EXECUTE_TEST. Function TESTS_COMPLETE returns a boolean value which indicates whether all the tests in the input file, "test_parameters" (described below), have been completed.

Procedure GET_TEST_PARAMETERS reads the criteria corresponding to a single test. The data is read from the user supplied input file "test_parameters". In test_parameters, the user provides the demonstrated "assurance" level (i.e., maximum probability of error) desired for the function and values of the Spec generic parameters to be used to instantiate the function, if it is generic. GET_TEST_PARAMETERS uses user supplied "GET" procedures, resident in IMPLEMENTATION, to read the generic value parameters. This is why Figure 3-1 shows that MAIN_PKG depends on IMPLEMENTATION. More detail is provided in Section VII of the user's manual (Appendix C).

Procedure EXECUTE_TEST executes a single test. It does so by renaming DRIVER to NEW_DRIVER in its declaration part and calling NEW_DRIVER in its body. The renaming is accomplished with an instantiation statement when generic parameters exist in the Spec or with a renaming statement otherwise. This technique was used because it provides a uniform interface with the DRIVER (i.e., NEW_DRIVER(assurance)). In addition, the renaming and instantiation statements are similar in structure and lend themselves well as alternative choices for code generation, whereas the choice between a procedure call in the body of the procedure and an instantiation statement with distinct procedure call are quite different and are more complex to generate mechanically.

3. DRIVER

DRIVER's architecture is shown in Figure 3-4. Notice that CHECK_PKG is renamed to BLACK_BOX and the actual function name is

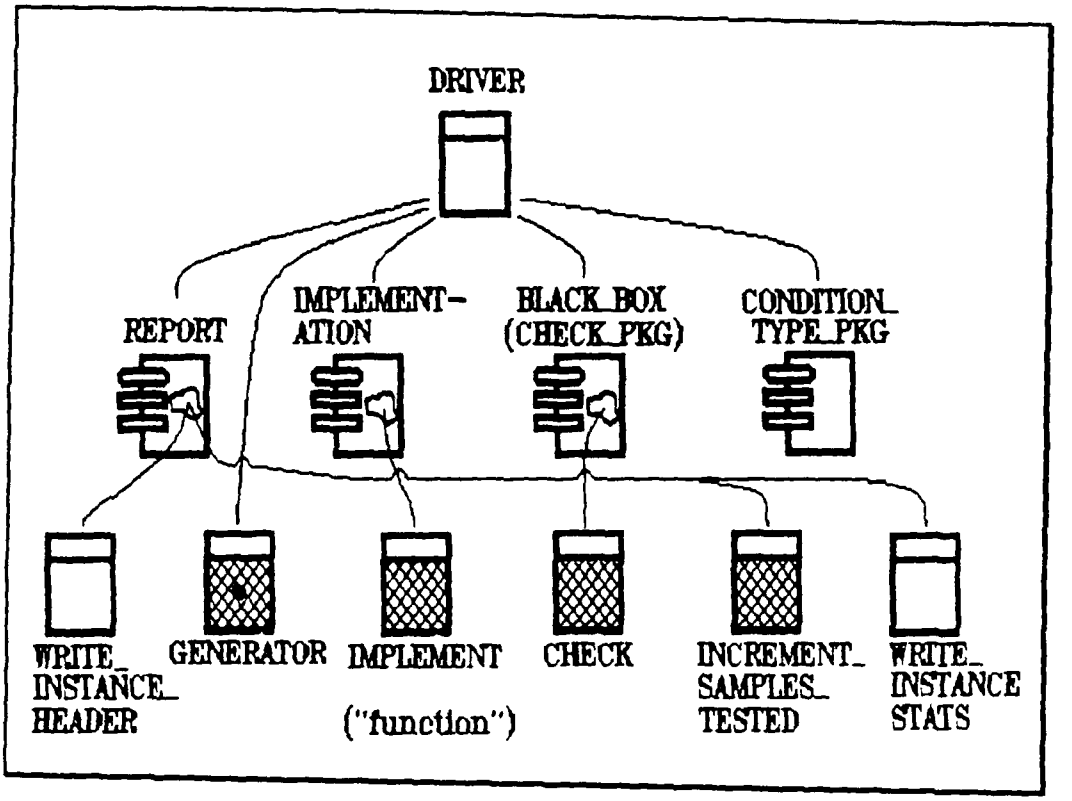

Figure 3-4. **DRIVER Architecture**

renamed to IMPLEMENT (as indicated by the parentheses ("( )")). A pruned version of DRIVER's data flow diagram is provided in Figure 3-5. Its interactions with MAIN_PKG and REPORT have been stripped out. Procedure DRIVER "drives" a single test: it repeatedly calls the function with message values provided by GENERATOR and sends the test set (reply values, message values, and "condition") to CHECK_PKG to be checked.

"Condition" is a variable, set in DRIVER, which explicitly indicates the termination condition of the function call. The function may terminate normally or raise an exception. To capture exceptions and set

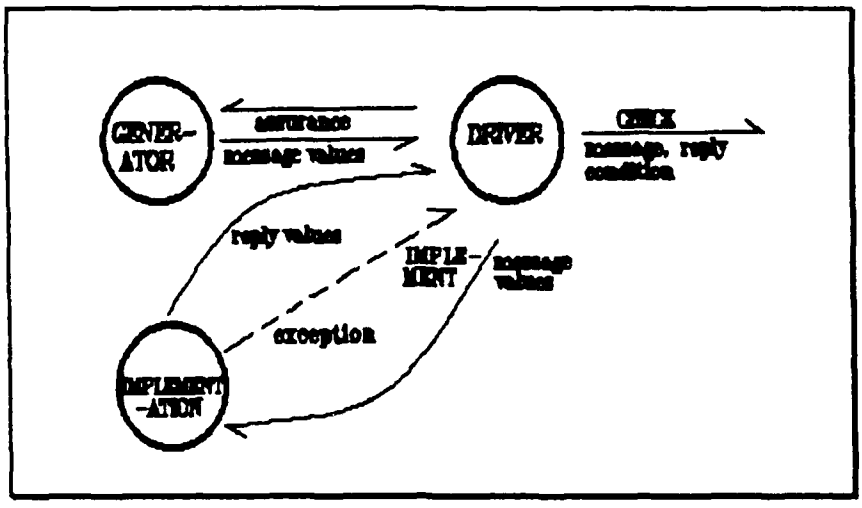

Figure 3-5. **DRIVER Data Flow Diagram**

the condition variable, the function call is embedded in an Ada block statement containing an exception handler. If the function call terminates normally, the "condition" variable will be set to "normal" and the exception handler is not invoked. However, if an exception is raised, control passes to the exception handler where the condition variable is set appropriately (see CONDITION_TYPE_PKG for more detail concerning "condition").

DRIVER is also responsible for incrementing sample counters maintained in REPORT after each test set is checked and for calling REPORT to write a "single test" header and summary as appropriate.

The design of DRIVER is based on the method for invoking generators presented in [Ref. 1:pp. 5-96–5-98]. It contains the "foreach" macro defined in Appendix E which is expanded before the component is compiled. An example is given in Figure 3-6. It should be read, "foreach 'x' generated by GENERATOR, execute the statements enclosed in the

last pair of brackets." The "assurance" is a parameter passed to the GENERATOR indicating the demonstrated "reliability" desired of the implementation (see GENERATOR for more details).

```
foreach([x:float], GENERATOR, [assurance], [
        **do these statements**
        ];)
```

Figure 3-6. **Sample "foreach" Macro**

4. CHECK_PKG

The data flow diagram of CHECK_PKG is provided in Figure 3-7. Package CHECK_PKG checks if a "REPLY" to a stimulus is correct. It has a single (visible) procedure CHECK. CHECK receives the test set (reply values, message values, condition) from DRIVER and checks that it is correct. If an error is found, it sends an error message to REPORT, otherwise it does nothing. CHECK_PKG uses subprocedures in MDOAG_LIB in the process of checking Specs.

CHECK_PKG is also based on the method for invoking generators mentioned in DRIVER. However, it will contain "foreach" macros only if the Spec contains QUANTIFIERS (discussed later). The "generators" used in connection with QUANTIFIERS are referred to as ITERATORS. Whether or not the Spec contains QUANTIFIERS, CHECK_PKG.M4 is generated and expanded to CHECK_PKG.A prior to Ada compilation.

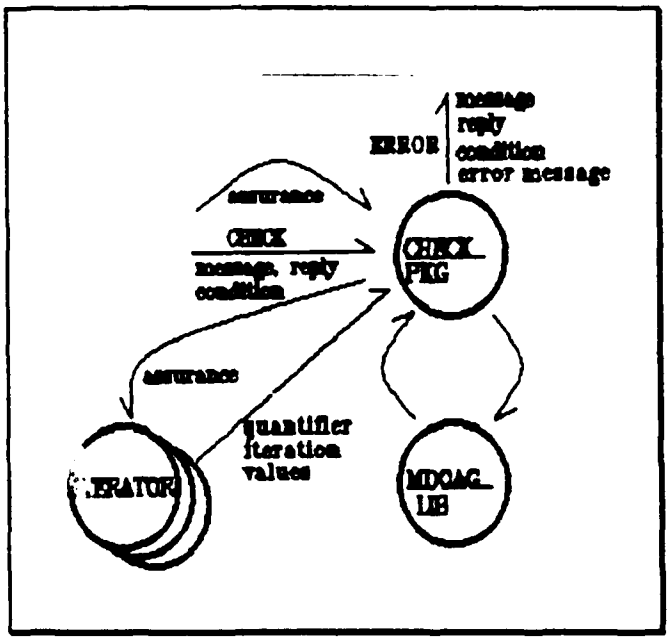

Figure 3-7. **CHECK_PKG Data Flow Diagram**

5. REPORT

Package REPORT is a machine that maintains statistical information (e.g., total tests, total samples tested, total errors, etc.) and manages the report (error file). All information in the report is generated by REPORT in response to requests by "using" modules. The services offered by REPORT are OPEN (create the error report and write the header), WRITE_INSTANCE_HEADER (write a header for a single test and initialize associated state variables), INCREMENT_SAMPLES_TESTED (increment samples counter state variables), ERROR (write an error to the report and increment error counter state variables), WRITE_INSTANCE_STATS (write a single test summary), and CLOSE (write a summary for all tests conducted and close the error file). Refer to Figure 3-2 to determine the users of the services.

6. CONDITION_TYPE_PKG

Package CONDITION_TYPE_PKG declares the type "CONDITION_TYPE" and provides its I/O subprocedures. CONDITION_TYPE is an enumerated data type representing the possible termination conditions of the function being tested. The normal termination condition is represented by the CONDITION_TYPE value "normal." There are two classes of exception conditions: specified exceptions and unspecified exceptions. Specified exceptions are those exceptions declared in the Spec specification. CONDITION_TYPE contains a unique value for each specified exception (i.e., the Spec exception name with the postfix "_condition"). Unspecified exceptions are Ada exceptions that may be raised during the function's execution but are not mentioned in the specification. The CONDITION_TYPE value "unspecified_exception" refers to all unspecified exceptions. Any occurence of an unspecified exception represents a program error.

7. GENERATOR

The procedure GENERATOR generates a sequence of message (input) values used by DRIVER in the conduct of a single test. The design of GENERATOR is based on the generator method presented in Reference 1 .

The system only provides a "generator" macro template which must be augmented with the Ada code necessary to generate message values. The sample in Figure 3-8 should be read,

> Generator "GENERATOR" receives a parameter "assurance"; it generates values of "x"; and the code required to generate those values is enclosed in the last pair of brackets ([ ]) in the macro definition.

The procedure 'generate' will be called with every "x" value generated.

```
generator(GENERATOR, [assurance: float], [x: float],
    [is
        n: constant natural :=
            natural((1.0/assurance)*float(BIN_LOG(1.0/assurance)));
        generated_x: float;
    begin
        for i in 1 .. n loop
            generated_x := RANDOM_NUMBER;
            generate(generated_x);
        end loop;
    end GENERATOR;])
```

Figure 3-8. **Sample Generator**

In the example, the the value "assurance" is an Ada float between 0.0 and 1.0 and corresponds to the measure of reliability desired of the function. This is the reciprocal of an upper bound on the mean interval between errors in an operational environment with the same distribution of input values as that provided by the test data generator, provided that the generated test set runs without detecting any errors. It is left to the user to implement the logic necessary to generate test input data with an appropriate distribution.

The macro expands into a generic procedure with a single generic procedure parameter which corresponds to the expanded "foreach" macro body of DRIVER. Instantiation of the GENERATOR amounts to passing in the "IMPLEMENTATION" and "CHECK" calls to be

executed once for each set of values generated. Refer to the user's manual (Appendix C) and the Samples of Appendix D for specific examples.

8. ITERATORS

An iterator procedure is partially generated for each quantifier expression contained in the Spec. Each iterator generates a sequence of values covering the range of values of the variables declared in the the quantifier. Like GENERATOR, the user supplies the iteration implementing code. Its design is identical to the GENERATOR except that the name of the GENERATOR corresponds to a particular Spec QUANTIFIER in the specification and its variables correspond to the variables declared in the QUANTIFIER.

9. IMPLEMENTATION

Package IMPLEMENTATION is user supplied. It holds four visible resources (or groups of resources) related to the implementation and required by other components of the MDOA:

1. Implementation of the Spec function to be tested.
2. Type declarations for types contained in the Spec.
3. I/O Routines for the abstract data types used in the Spec.
4. Declarations of exceptions contained in the Spec.

The Spec function is implemented in accordance with Spec concrete interface conventions. The user supplies the declarations of all data types used in the Spec (that are not contained in Ada STANDARD) and I/O routines for those types used by the MDOA (i.e., MAIN_PKG and REPORT). The exceptions are declared in the visible portion of the package to provide the DRIVER the capability to capture the exception and set

the condition. Refer to the user's manual (Appendix C) and the Samples of Appendix D for more details.

10. MDOAG_LIB

Package MDOAG_LIB contains utilities used by the MDOA. Currently, it contains only two services: functions implementing the Spec "if and only if" and "implies" operators.

IV. IMPLEMENTATION OF THE MDOAG

This chapter describes the template translation methodology used in the translation process and provides a detailed description of the templates used to implement each generated component of the Module Driver and Output Analyzer.

A. TRANSLATION TEMPLATE METHODOLOGY

A simple scheme presented in Reference 15 was chosen to automatically translate arbitrary specifications in Spec into the architecture's Ada components. For each component in the architecture, a translation "template" was developed into which specification dependent portions of code could be inserted.

Obviously, the individual components that are generated are dependent upon the specification (e.g., names, number of arguments, assertions, concepts, etc.) read by the Module Driver and Output Analyzer Generator. However, each of the individual components has a fixed part, a "template" or "shell," that does not vary with the specification (e.g., package/subprogram name, some package dependencies (with/use), etc.). The fixed and variable portions of each of the architecture's components were determined, resulting in the translation templates contained in Appendix F.

A graphic representation of the template process is provided in Figures 4-1 through 4-3. Figure 4-1 shows an abstract syntax tree for a

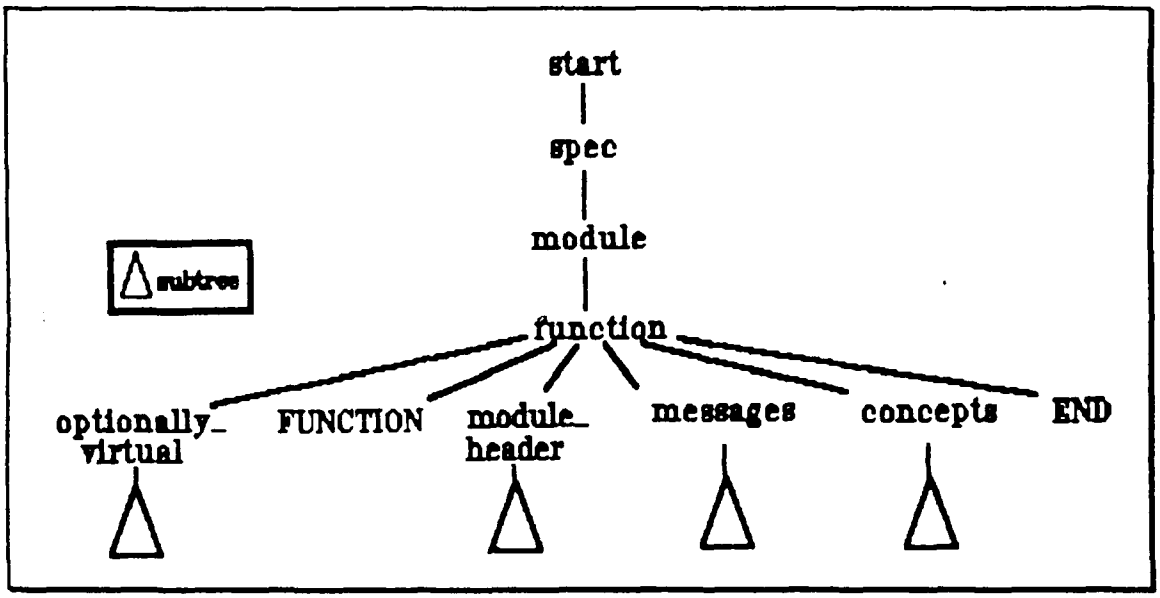

Figure 4-1. **Graphical Representation of an Abstract Syntax Tree for a Spec**

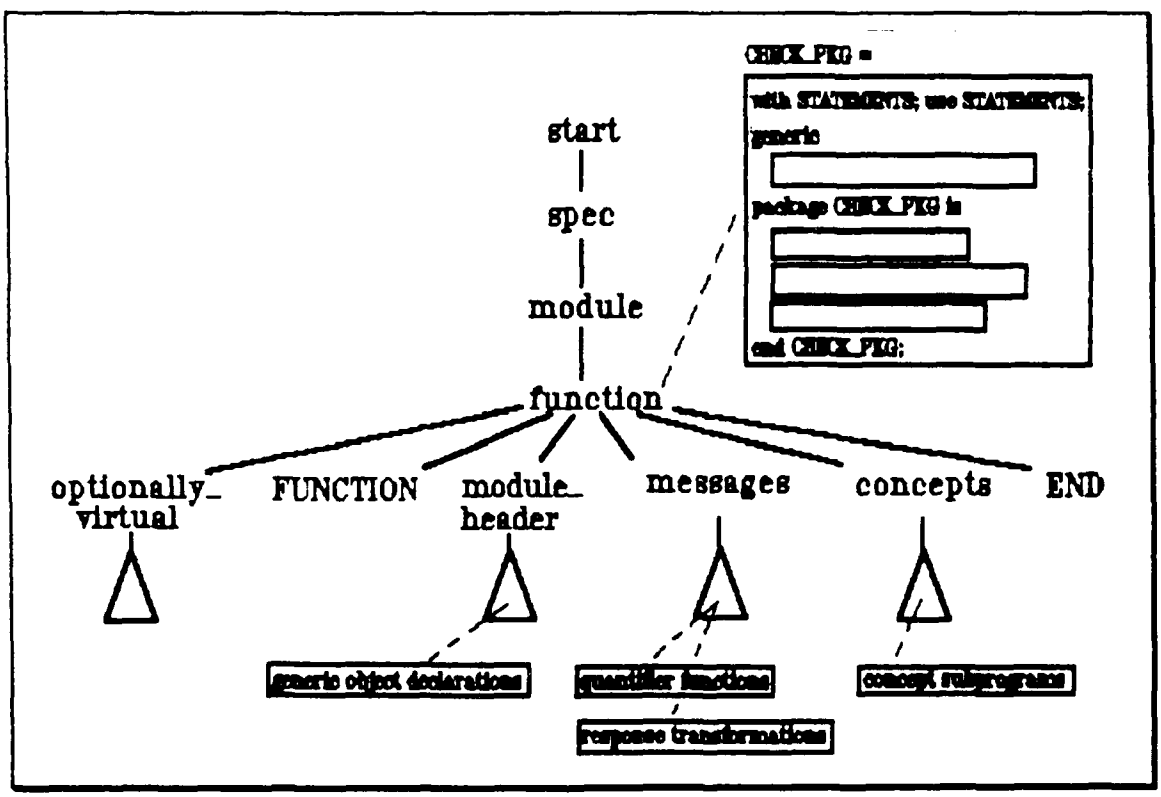

Figure 4-2. **Template Association to the Abstract Syntax Tree of the Spec**

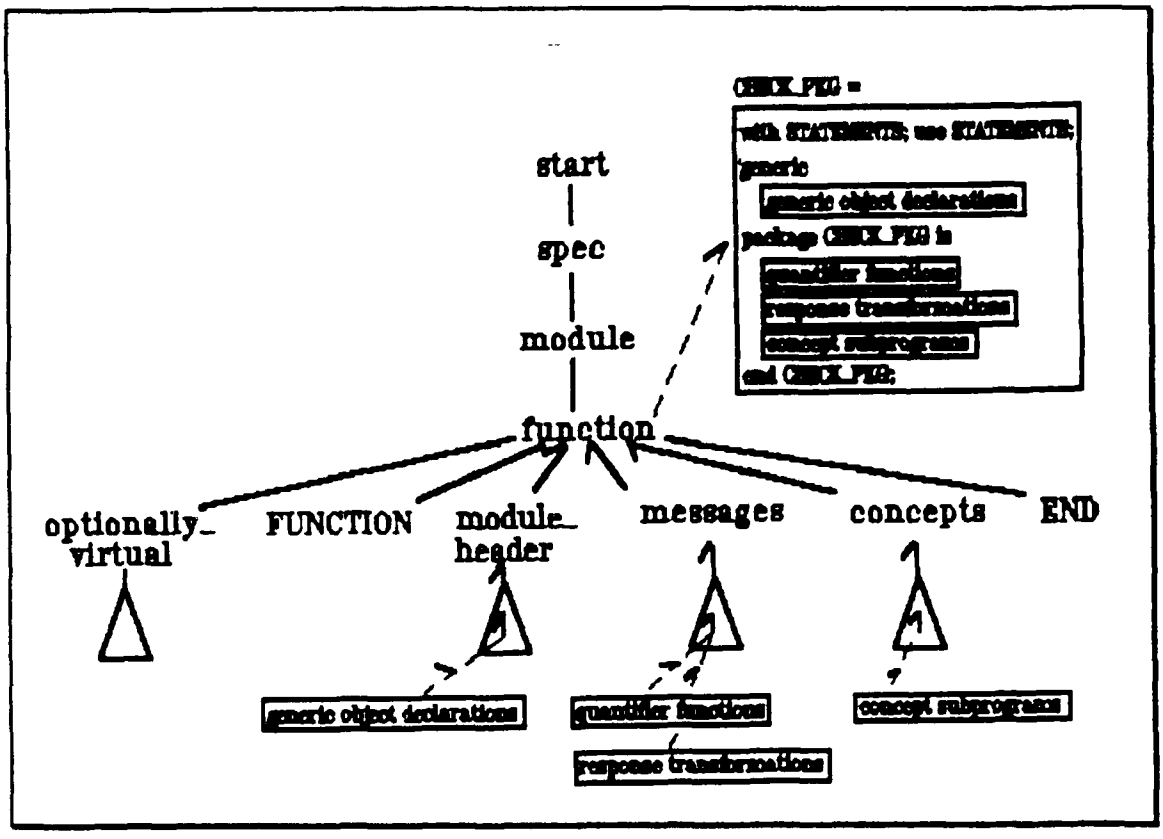

Figure 4-3. **Template Completion Process for the Spec**

subset of Spec. A complete depiction of the entire tree, down to the terminal symbols, is not possible because most trees are very large. Instead, triangles are used to represent the subtrees of the non-terminal to which it is attached.

Figure 4-2 shows how templates are associated with particular nodes of the tree. In this case, the CHECK_PKG template (an attribute) is "attached" to the function non-terminal symbol. It has four missing elements, or slots, which must be filled in (i.e., Spec-dependent variable information).

The missing information corresponds to "generic object declarations," "quantifier functions," "response transformations" and "concept subprograms." That information is found in the "module_header," "messages," and "concepts" subtrees, as depicted by the "subtemplates" attached to the subtrees. The figure indicates that attribute evaluation is complete for those subtemplates because they are filled in. (Note: The subtemplates themselves may have subtemplates.)

Figure 4-3 shows how the CHECK_PKG template is completed. Arrowheads have been added to the arcs of the tree to depict the flow of information. Each subtemplate moves up the tree, via its subtree, and is inserted into its appropriate slot in the CHECK_PKG template. The flow of information is accomplished with semantic functions that simply pass the information from node to node.

Once the CHECK_PKG template is completely filled in, it is passed up the tree to the "start" symbol where it is output to a file. To determine the component file name, examine the "%outfile" semantic function associated with the start symbol in Section 3 of Appendix A.

Use of the template methodology does not restrict information flow to the upward direction. It is necessary to pass information in all directions. For instance, it is often necessary to use information contained in an subtemplate (or attribute) of one subtree to complete a subtemplate of a distant subtree before the "distant" subtemplate may be passed up the tree. The figures present a simplified but valid view of the process. In addition, the CHECK_PKG is not an accurate version of the template used in the code. It was simplified for the sake of illustration. Refer to the

translation templates of Appendix F for the templates actually used in the system.

B. MAIN_PKG TEMPLATE

The MAIN_PKG template is given in Figure 4-4. It has three variable portions (slots) to be filled in:

1. **GENERIC OBJECT DECLARATIONS**
2. **GENERIC OBJECT GETS**
3. **DRIVER INSTANTIATION OR RENAMING DECLARATION**

A "pruned" version of the resulting code generated for the "generic square root" example of Figure 1-4 is presented in Figure 4-5.

1. **GENERIC OBJECT DECLARATIONS**

The purpose of the **GENERIC OBJECT DECLARATIONS** is to generate "object declarations" for variables containing values for generic parameters (i.e., not "generic" object declarations). The variables are used as actual parameters in the instantiation of DRIVER, which subsequently uses their values to instantiate the function. One object declaration is generated per generic parameter in the Spec and derives its name directly from the corresponding Spec NAME.

The **GENERIC OBJECT DECLARATIONS** generated for the generic square root example is the single Ada declaration of "precision," the lone generic parameter ("precision") of the generic square root Spec (line *1*, Figure 4-5). If the Spec had more generic parameters, they would follow in suit. If the Spec had no generic parameters, nothing (i.e., empty string) would be generated.

```
package MAIN_PKG is
     function TESTS_COMPLETE return boolean;
     procedure GET_TEST_PARAMETERS;
     procedure EXECUTE_TEST;
end MAIN_PKG;

with FLT_IO;
with DRIVER;
with IMPLEMENTATION;
with TEXT_IO; use TEXT_IO;
package body MAIN_PKG is
     INFILE: FILE_TYPE;
     ASSURANCE: FLOAT range 0.0..1.0;
     **GENERIC OBJECT DECLARATIONS**                              *1*

     function TESTS_COMPLETE return boolean is
     begin
          if IS_OPEN(INFILE) and then END_OF_FILE(INFILE) then
               CLOSE(INFILE);
               return TRUE;
          elsif IS_OPEN(INFILE) then
               return FALSE;
          else OPEN(INFILE,IN_FILE,"test_parameters");
               return END_OF_FILE(INFILE);
          end if;
     end TESTS_COMPLETE;

     procedure GET_TEST_PARAMETERS is
     begin
          FLT_IO.GET(INF   E,ASSURANCE);
          **GENERIC OBJECT GETS**                                 *2*
     end GET_TEST_PARAMETERS;

     procedure EXECUTE_TEST is
          **DRIVER INSTANTIATION OR RENAMING DECLARATION**        *3*
     begin
          NEW_DRIVER(ASSURANCE);
     end EXECUTE_TEST;
end MAIN_PKG;
```

Figure 4-4. **MAIN_PKG Template**

```
package MAIN_PKG is
        function TESTS_COMPLETE return boolean;
        procedure GET_TEST_PARAMETERS;
        procedure EXECUTE_TEST;
end MAIN_PKG;

with FLT_IO;
with DRIVER;
with IMPLEMENTATION;
with TEXT_IO; use TEXT_IO;
package body MAIN_PKG is
    INFILE: FILE_TYPE;
    ASSURANCE: FLOAT range 0.0..1.0;
    precision: float;                                          *1*
                .
                .
                .

    procedure GET_TEST_PARAMETERS is
    begin
        FLT_IO.GET(INFILE,ASSURANCE);
        IMPLEMENTATION.GET(INFILE,precision);                  *2*
    end GET_TEST_PARAMETERS;

    procedure EXECUTE_TEST is
        procedure NEW_DRIVER is new DRIVER(precision);         *3*
    begin
        NEW_DRIVER(ASSURANCE);
    end EXECUTE_TEST;
end MAIN_PKG;
```

Figure 4-5. **MAIN_PKG of the Generic Square Root Spec of Figure 1-4**

Generating object declarations, like a host of other "declaration-like" statements (parameter specification, generator loop variables, etc.), is simply the concatenation of the tokens required by the statement. All the required tokens are located within the field_list subtree. Although a trivial generation, several methods can be used to implement the

generation and not all schemes are as useful as others. The method preferred in this research is to generate all declarations as individual object declarations rather than as multiple object declarations. The two forms are equivalent [Ref. 2:p. 3-3]. In this method, single declarations (less trailing delimiters) are formed at the lowest subtree possible (i.e., the name list subtree) and built into lists by passing them up the tree (i.e., name_list to type_ binding to field_list), adding delimiters as required. The same technique is used to generate the generator loop variables discussed later (e.g., See "gen_loop_vars" attribute at the name_list, type_binding, field_list of Appendix A). The advantage of this technique is it can be applied to all "declaration-like" statements without loss of functionality while allowing manipulation of the individual elements of the list (e.g., removal of items, additions, replacements, etc.).

2. **GENERIC OBJECT GETS**

The purpose of **GENERIC OBJECT GETS** is to generate the calls to input procedures required to read the generic objects for instantiating generic Spec functions. This is needed because generic functions must be instantiated before they can be tested. The particular instances to be tested are specified by the user via a file containing actual values for the generic parameters.

The **GENERIC OBJECT GETS** generated for the generic square root function is statement on line *2*, Figure 4-5. If more generic parameters existed, a "GET" statement would have been generated for each generic object.

The generation of individual statements is accomplished at the name_list subtree (See the "mpkg_gets" attribute, Appendix A). Note that the definition of the "GET" procedure is supplied by the implementor in package IMPLEMENTATION (i.e., IMPLEMENTATION.GET(...)). It is translated this way to avoid errors resulting from overloading the subprocedure GET. Additionally, the generic parameter may be of an "implementor"-defined type, unique to the implementation, rather than a standard Ada type. For instance, the Spec could have defined "precision" to be type "real," for which no standard Ada type exists. Consequently, the system requires that the implementor provide the input procedure. (More information on the "GET" procedure can be found in the user's manual, Appendix C).

3. **DRIVER INSTANTIATION OR RENAMING DECLARATION**

The purpose the **DRIVER INSTANTIATION OR RENAMING DECLARATION** is to instantiate or rename the DRIVER as appropriate. The DRIVER procedure is generic only when the Spec function is generic.

The **DRIVER INSTANTIATION OR RENAMING DECLARATION** generated for generic square root is line *3*, Figure 4-5. In this case, the Spec is generic. If the Spec had more generic parameters, they would have been included in the instantiation. If the Spec had no generic parameters, the non-generic DRIVER procedure would have been renamed as follows: "procedure NEW_DRIVER(assurance: float) renames DRIVER;."

Generation of this slot is accomplished using a subtemplate located at the function non-terminal (See the driver_basic_decl attribute,

function non-terminal, Appendix A and MAIN_PKG subtemplate Appendix F).

C. DRIVER TEMPLATE

The DRIVER template is given in Figure 4-6. It has seven slots to be filled in:

1. **GENERIC FORMAL PART**
2. **PARAMETER SPECIFICATIONS**
3. **INSTANTIATION OR RENAMING DECLARATIONS**
4. **GENERATOR LOOP VARIABLES**
5. **FUNCTION CALL**
6. **EXCEPTION WHEN CLAUSES**
7. **FORMAL MESSAGE ACTUAL PARMS**

Slots one through seven are located on lines *1* through *7*, respectively.

A copy of the macro file "generator.m4" is included in Appendix E, and an electronic copy is available in suns2:/work/student/depasqua/MACROS.

The DRIVER generated for the generic square root example is presented in Figure 4-7.

1. **GENERIC FORMAL PART**

The purpose of **GENERIC FORMAL PART** is to generate the specification of the objects that represent the Spec formal arguments, as

```
include(/n/suns2/work/student/depasqua/MACROS/generator.m4)
**GENERIC FORMAL PART**                                          *1*
procedure DRIVER(assurance: in FLOAT);

with GENERATOR;
with CHECK_PKG;
with REPORT; use REPORT;
with IMPLEMENTATION; use IMPLEMENTATION;
with CONDITION_TYPE_PKG; USE CONDITION_TYPE_PKG;

procedure DRIVER is (assurance: in float) is
    condition: condition_type := normal;
    **PARAMETER SPECIFICATIONS**                                 *2*
    **INSTANTIATIONS OR RENAMING DECLARATIONS**                  *3*
begin
    REPORT.WRITE_INSTANCE_HEADER;
    foreach[[(**GENERATOR LOOP VARIABLES**), GENERATOR,          *4*
            [(assurance)], [
        begin
            **FUNCTION CALL**                                    *5*
            condition := normal;
        exception
            **EXCEPTION_WHEN_CLAUSES**                           *6*
            when others =>
                condition := unspecified_exception;
        end;
        BLACK_BOX.CHECK(condition,
                **FORMAL MESSAGE ACTUAL PARMS**);                *7*
        INCREMENT_SAMPLES_TESTED;])
    REPORT.WRITE_INSTANCE_STATS;
end DRIVER;
```

Figure 4-6. **DRIVER Template**

```
include(/n/suns2/work/student/depasqua/MACROS/generator.m4)
generic                                                          *1*
    precision: float;
procedure DRIVER(assurance: in float);

with GENERATOR;
with CHECK_PKG;
with REPORT; use REPORT;
with IMPLEMENTATION; use IMPLEMENTATION;
with CONDITION_TYPE_PKG; use CONDITION_TYPE_PKG;

procedure DRIVER(assurance: in float) is
    condition: condition_type := normal;
    x: float;                                                    *2*
    y: float;
    function IMPLEMENT is new square_root(precision);            *3*
    package BLACK_BOX is new CHECK_PKG (precision, assurance);
begin
    REPORT.WRITE_INSTANCE_HEADER;
    foreach([x:float], GENERATOR.[assurance], [    *4*
        begin
            y := IMPLEMENT(x);                                   *5*
            condition := normal;
        exception
            when imaginary_square_root =>                        *6*
                condition := imaginary_square_root_condition;
            when others =>
                condition := unspecified_exception;
        end;
        BLACK_BOX.CHECK(condition, x, y);                        *7*
        INCREMENT_SAMPLES_TESTED;])
    REPORT.WRITE_INSTANCE_STATS;
end DRIVER;
```

Figure 4-7. **DRIVER of the Generic Square Root Spec of Figure 1-4**

the Ada generic formal part of the DRIVER. When the Spec is generic, a **GENERIC FORMAL PART** is generated; otherwise, an empty string is generated resulting in a non-generic DRIVER procedure.

The generic part generated for the generic square root example is "generic" followed by the generic parameter declaration of "precision" (line *1*, Figure 4-7).

See the function.g_formal_part semantic function at function, Appendix A, for the simple translation.

2. **PARAMETER SPECIFICATIONS**

The purpose of **PARAMETER SPECIFICATIONS** is to generate declarations for variables that represent the formal arguments declared in the Spec "MESSAGE" and "REPLY."

In the square root example, the variables "x" and "y" of type "float" are specified (line *2*, Figure 4-6). The Spec argument names are used verbatim.

The generation is accomplished through the simple concatenation of message and response parameter specifications (See the message.parm_specs semantic function at message, Appendix A).

3. **INSTANTIATIONS OR RENAMING DECLARATIONS**

The purpose of the **INSTANTIATIONS OR RENAMING DECLARATIONS** slot is to:

1. Effect the proper concrete interface to the function's Ada implementation.
2. Rename the function to the standard name "IMPLEMENT."
3. Instantiate the generic package CHECK_PKG.
4. Rename CHECK_PKG to "BLACK_BOX."

The four possible translations are shown in Figure 4-8. The translation generated depends on the concrete interface called for by the Spec. The four Ada interfaces supported are:

1. Generic function.
2. Non-generic function.
3. Generic procedure.
4. Non-generic procedure.

The Spec language has definitive concrete interface generation rules for Ada [Ref. 1:pp. 4-54 - 4-55]:

1. A message with a reply with a single data component corresponds to an Ada function with an "in" parameter for each component of the MESSAGE and a "return" corresponding to the single data component of the REPLY.
2. A message with a GENERATE corresponds to a generic procedure parameter. (This interface is not supported by the current version of MDOAG.)
3. A message that does not fall into the categories above, corresponds to an Ada procedure with an "in" parameter for each component of the MESSAGE and an "out" for each component of the REPLY if there is one.

The rules above may be modified using Spec PRAGMAs. One such modification is supported. PRAGMA "update(x,y)" indicates that the "x" component of a MESSAGE and the "y" component of a REPLY both correspond to the same "in out" parameter of the Ada subprogram with the formal parameter name "x." [Ref. 1:pp. 4-54–4-55]

Although not explicitly stated in the concrete interface rules, it is clear that a generic specification requires that the rules above be followed and that the appropriate generic part be added to the interface.

```
**INSTANTIATIONS OR RENAMING DECLARATIONS**
--Generations based on the type of Ada interface called for by
--the Spec and whether or not the function is generic.

--non-generic function:
function IMPLEMENT(**FORMAL MESSAGE PARM SPECIFICATIONS**)
        return **TYPE MARK** renames **FUNCTION DESIGNATOR**;
package BLACK_BOX is new CHECK_PKG(assurance);

--non-generic procedure:
procedure IMPLEMENT(**FUNCTION CALL SPECIFICATIONS**)
            renames IMPLEMENTATION.**FUNCTION DESIGNATOR**;
package BLACK_BOX is new CHECK_PKG(assurance);

--generic function:
function IMPLEMENT is new
    **FUNCTION DESIGNATOR**(**GENERIC ACTUAL PARAMETERS**);
package BLACK_BOX is new CHECK_PKG (**GENERIC ACTUAL
                                PARMS**, assurance);

generic procedure:
procedure IMPLEMENT is new
    **FUNCTION DESIGNATOR**(**GENERIC ACTUAL PARAMETERS**);
package BLACK_BOX is new CHECK_PKG (**GENERIC ACTUAL
                                PARMS**, assurance);
```

Figure 4-8. **DRIVER Template **INSTANTIATIONS OR RENAMING DECLARATIONS****

Based on the concrete interface generation rules and the fact that the "generic square root" Spec is generic, the "generic function" **INSTANTIATIONS OR RENAMING DECLARATIONS** translation of Figure 4-8 is generated (line *3*, and the following statement, Figure 4-6).

Another example is useful to demonstrate the alternate interface generation option. Assume the square root function is specified as in Figure 4-9. It is non-generic (line *1*, Figure 4-9) and contains an

"update" PRAGMA (line *2*, Figure 4-9). In this case, the Module Driver and Output Analyzer Generator generates the DRIVER of Figure 4-10. The interface rules mandate that the Spec function be implemented as a non-generic procedure with a single "in out" parameter "x." Consequently, the "non-generic procedure" translation of Figure 4-8 is generated with the appropriate "in out" declaration (line *3* and the following statement of Figure 4-10).

```
FUNCTION square_root          *1*
        MESSAGE(x: float)
            PRAGMA update(x,y)                                    *2*
            WHEN x >= 0.0
                REPLY(y: float)
                WHERE y > 0.0, approximates(y * y, x)
            OTHERWISE REPLY EXCEPTION imaginary_square_root
        CONCEPT approximates(r1 r2: float)
            VALUE(b: boolean)
                WHERE b <=> abs(r1 - r2) <= abs(r2 * 0.001)
END
```

Figure 4-9. **Non-Generic "square_root" With "update" PRAGMA**

Although better handled as an inspection or static analysis test item, the instantiation and renaming declarations provides a crude "test" for proper interface implementation. An improper interface will result in an error message from the Ada compiler.

```
include(/n/suns2/work/student/depasqua/MACROS/generator.m4) *1*

procedure DRIVER(assurance: in float);

with GENERATOR;
with CHECK_PKG;
with REPORT; use REPORT;
with IMPLEMENTATION; use IMPLEMENTATION;
with CONDITION_TYPE_PKG; use CONDITION_TYPE_PKG;

procedure DRIVER(assurance: in float) is
    condition: condition_type := normal;
    x: float;                                                    *2*
    y: float;
    procedure IMPLEMENT(x: in out float) renames                 *3*
            IMPLEMENTATION.square_root;
    package BLACK_BOX is new CHECK_PKG(assurance);
begin
    REPORT.WRITE_INSTANCE_HEADER;
    foreach([x:float], GENERATOR,[assurance], [                  *4*
        begin
            y := x;                                              *5*
            IMPLEMENT(y);
            condition := normal;
        exception
            when imaginary_square_root =>                        *6*
                condition := imaginary_square_root_condition;
            when others =>
                condition := unspecified_exception;
        end;
        BLACK_BOX.CHECK(condition, x, y);                        *7*
        INCREMENT_SAMPLES_TESTED;])
    REPORT.WRITE_INSTANCE_STATS;
end DRIVER;
```

Figure 4-10. **DRIVER for Non-Generic "square_root" With "update" PRAGMA**

The generation of **INSTANTIATIONS OR RENAMING DECLARATIONS** is accomplished by:

1. Recognizing that the interface type is "function" only when there is one response parameter and no update pragmas in the Spec; otherwise, it is "procedure."

2. Declaring two attributes and writing semantic rules to count the response parameters and updates. (See the attributes "update_ court" at pragmas and "r_parm_count" at name_list, type_binding, field list, response_set and response_cases, Appendix A.)

3. Using the totals to determine the interface type. (See the attribute ada_ interface_type at function, Appendix A.)

4. Declaring and creating a map, "update," which maps the message-parameters-being-updated to the reply parameters of the update pragmas (See "update" at pragmas, actuals Appendix A) and using that map to determine the proper parameter mode (i.e., "in out" for updated parameter) of the object specifications used in the function renaming. (See the attribute "fm_call_specs" at name_list, Appendix A.)

5. Declaring and creating a map, "remove," which maps reply-parameters-updating-message-parameters to "true" (See the attribute "remove" at pragmas, actuals, Appendix A) and using the map to remove reply parameters from the object specifications used in the function renaming (See the attribute "r_call_specs" at name_list.).

4. **GENERATOR LOOP VARIABLES**

The purpose of the **GENERATOR LOOP VARIABLES** slot is to generate the parameter specifications for the MESSAGE (input) variables that will be generated by the GENERATOR and used in the function call. In effect, they are the formal arguments of the Spec message formatted for the generator.m4 macro.

The **GENERATOR LOOP VARIABLES** for the generic square root example is "x:float," (line *4* Figure 4-6), because "x" is the only MESSAGE parameter. Additional parameters would follow in suit.

Generation is straightforward. Refer to the discussion on generating **GENERIC OBJECT DECLARATIONS** in MAIN_PKG, Chapter IV.

5. **FUNCTION CALL**

The purpose of the **FUNCTION CALL** is generate the function call and to ensure that all the variables representing the Spec MESSAGE and REPLY parameters are appropriately set prior to checking the results of the function call. When the concrete interface of the Spec function is an Ada function, the function call accomplishes both purposes. When the Spec function is coded as an Ada procedure due to an "update" progma, the REPLY parameters must be initialized to the MESSAGE variables they "update" prior to making the procedure call.

The generic square root **FUNCTION CALL** (line *5* of Figure 4-6) illustrates the simple Ada function call. The variables "x" and "y" are properly set by the function call to allow the results to be checked by procedure CHECK.

The non-generic square root **FUNCTION CALL** (line *5* and the following statement of Figure 4-10) illustrates the translation when the Spec function is implemented as an Ada procedure due to a Spec "update" pragma. The REPLY parameter "y" is set to "x," the MESSAGE value provided by the GENERATOR. Then "y" is used in the procedure call. In this way, the value of "x" is preserved and "y" receives the REPLY of the function. (The parameter mode of the procedure is "in out" (line *3*, Figure 4-10).) The variables "x" and "y" are properly set to allow the results to be checked by procedure CHECK.

The generation is accomplished by:

1. Basing the generation on the ada interface type (See the attribute "call" at function, Appendix A).

2. Generating initializing statements from the update pragmas and passing them up the tree for use in the interface. (See the attribute "init_statements" at pragma, Appendix A.)

3. Using the "update" map to "replace" the message parameter with the reply parameter as the actual parameter in the procedure call. (See the attribute "fn_call_actuals" at name_list, Appendix A.)

4. Using the "remove" map to remove reply parameters from the actual parameters of the function call. (See the attribute "r_call_actuals" at name_list, Appendix A.)

6. **EXCEPTION WHEN CLAUSES**

The purpose of the **EXCEPTION WHEN CLAUSES** slot, line *6* Figure 4-6, is to generate a simple sequence of zero or more Ada exception handlers, one for each exception enumerated in the Spec. Each handler simply sets the condition variable to identify the exception that occurred.

The **EXCEPTION WHEN CLAUSES** generated for the square root example is located on line *6* of Figure 4-10.

Generation is straightforward. (See the DRIVER subtemplates, Appendix F.)

7. **FORMAL MESSAGE ACTUAL PARMS**

The purpose of **FORMAL MESSAGE ACTUAL PARMS** slot, line *7* Figure 4-6, to generate the list of actual parameters representing the Spec MESSAGE and REPLY values to be checked by procedure CHECK.

The **FORMAL MESSAGE ACTUAL PARMS** generated for the generic square root example is shown on line *7* Figure 4-10.

The generation is straightforward.

D. CHECK_PKG TEMPLATE

The CHECK_PKG template is presented in Figure 4-11. It has eight slots to be filled in:

1. **GENERIC OBJECT DECLARATIONS**
2. **PARAMETER SPECIFICATIONS**
3. **QUANTIFIER WITH STATEMENTS**
4. **CONCEPT SUBPROGRAM SPECIFICATIONS**
5. **PARAMETER SPECIFICATIONS** (duplicate of 2)
6. **QUANTIFIER FUNCTIONS**
7. **RESPONSE TRANSFORMATION**
8. **CONCEPT SUBPROGRAM BODIES**

Slots one through eight are located on lines *1* through *8*, respectively.

The structure the template attempts to preserve the basic format of the Spec. Following the basic Spec format makes it easier to modify the translation if desired. It also makes it easier to visually inspect that the generated code is correct. The Spec format could not always be followed precisely. For instance, there are inconsistencies between Spec and Ada visibility rules. Consequently, to adhere to Ada visibility rules **CONCEPT SUBPROGRAM SPECIFICATIONS** had to precede **RESPONSE TRANSFORMATION**, which depend on the former. Spec

```
include(/n/suns2/work/student/depasqua/MACROS/generator.m4)
with REPORT; use REPORT;
with IMPLEMENTATION; use IMPLEMENTATION;
with CONDITION_TYPE_PKG; use CONDITION_TYPE_PKG;
generic
     assurance: float;
     **GENERIC OBJECT DECLARATIONS**                              *1*
package CHECK_PKG is
     procedure CHECK(condition: condition_type;
                     **PARAMETER SPECIFICATIONS**);               *2*
end CHECK_PKG;

**QUANTIFIER WITH CLAUSES**                                       *3*
package body CHECK_PKG is

     **CONCEPT SUBPROGRAM SPECIFICATIONS**                        *4*

     procedure CHECK(condition: condition_type;
                     *PARAMETER SPECIFICATIONS**) is              *5*
          preconditions_satisfied: boolean := false;

          **QUANTIFIER FUNCTIONS**                                *6*
     begin
          **RESPONSE TRANSFORMATION**                             *7*
     end CHECK;

     **CONCEPT SUBPROGRAM BODIES**                                *8*

end CHECK_PKG;
```

Figure 4-11. **CHECK_PKG Template**

CONCEPTs are visible throughout the entire function [Ref. 1:p. 3-102], even though they are enumerated last in the specification. QUANTIFIER evaluating functions had to be placed out-of-format.

The CHECK_PKG generated for the generic square root example of Figure 1-4 is presented in Figure 4-12.

```
include(/n/suns2/work/student/depasqua/MACROS/generator.m4)
with REPORT; use REPORT;
with MDOAG_LIB; use MDOAG_LIB;
with IMPLEMENTATION; use IMPLEMENTATION;
with CONDITION_TYPE_PKG; use CONDITION_TYPE_PKG;
generic
      assurance: float;
      precision: float;                                                    *1*
package CHECK_PKG is
      procedure CHECK(condition: condition_type;
                                      x: float;                            *2*
                                      y: float);
end CHECK_PKG;
                                                                           *3*
package body CHECK_PKG is

      function approximates(r1, r2: float) return boolean;                 *4*

      procedure CHECK(condition: condition_type;
                                      x: float;                            *5*
                                      y: float) is
            preconditions_satisfied: boolean := false;
                                                                           *6*
      begin
            if (x >= 0.0) then                                             *7*
                  if not ((y > 0.0)) then                                  *8*
                        REPORT.ERROR(condition, x, y,
                              "WHEN x>=0.0 NOT y>0.0");
                  end if;
                  if not (approximates((y * y) , x)) then                  *9*
                        REPORT.ERROR(condition, x, y,
                              "WHEN x>=0.0 NOT approximates(y * y,x)");
                  end if;
                  preconditions_satisfied := true;
            end if;
            if not (precondition_satisfied) then
                  if not (condition =
                              imaginary_square_root_condition) then
                        REPORT.ERROR(condition, x, y,
                  "OTHERWISE NOT EXCEPTION imaginary_square_root");
            end if;
      end if;
end CHECK;

      function approximates(r1, r2: float) return boolean is               *10*
            b: boolean;
      begin
            return(abs((r1 - r2)) <= abs((r2 * precision)));               *11*
      end approximates;
end CHECK_PKG;
```

Figure 4-12. **CHECK_PKG of the Generic Square Root Spec of Figure 1-4**

1. ****GENERIC OBJECT DECLARATIONS****

The ****GENERIC OBJECT DECLARATIONS**** slot is identical to MAIN_PKG ****GENERIC OBJECT DECLARATIONS****. Refer to Section B of this chapter.

2. ****PARAMETER SPECIFICATIONS****

The ****PARAMETER SPECIFICATIONS**** is identical to DRIVER ****PARAMETER SPECIFICATIONS****. Refer to Section C o this chapter.

3. ****QUANTIFIER WITH CLAUSES****

The purpose of ****QUANTIFIER WITH CLAUSES**** is to introduce ITERATORS to CHECK_PKG by generating a "with ****ITERATOR NAME**;" for each QUANTIFIER in the Spec. ITERATORS are modules used to generate the values in the range of a QUANTIFIER,to check Specs containing QUANTIFIERS.

The generic square root example has no QUANTIFIERS, resulting in the generation of an empty string for this slot (line *3*, Figure 4-12).

Generation of a single "with" statement is straightforward. (See "q_with_clauses" at expression -> quantifier, Appendix A.). The mechanism used to collect the clauses from expression lists and expressions is analogous to the method used to collect quantifier functions presented in Section D of this chapter.

4. ****CONCEPT SUBPROGRAM SPECIFICATIONS****

The purpose of ****CONCEPT SUBPROGRAM SPECIFICATIONS**** is to provide package-wide visibility of "implemented" CONCEPTs. CONCEPTs are implemented as Ada functions. ****CONCEPT SUBPROGRAM

SPECIFICATIONS** are Ada function specifications for those CONCEPTS. One specification is generated for each CONCEPT contained in the Spec.

The CONCEPT "approximates," of the generic square root Spec, results in the generation of line *4* Figure 4-12. Its location at the top of the package body provides visibility for its use in procedure CHECK without greatly deviating from the format of the Spec.

Generation and collection of CONCEPT specifications is straightforward (See the attribute "c_subprog_spec" at concept, Appendix A).

5. **QUANTIFIER FUNCTIONS**

The purpose of **QUANTIFIER FUNCTIONS** is to evaluate QUANTIFIER expressions (e.g., ALL, SOME, etc.) contained in the Spec and return the result. The **QUANTIFIER FUNCTIONS** slot is filled with zero or more Ada functions and associated parameter specifications.

The generic square root example contains no quantifiers, consequently the empty string is generated (line *6*, Figure 4-12). The function "with_quantifiers" of Figure 4-13 contains a postcondition specified with the ALL and SOME QUANTIFIERS (i.e., universal and existential

```
FUNCTION with_quantifiers
    MESSAGE(x y: type_1)
        REPLY(z: type_1)
        WHERE ALL(q: type_1 SUCH THAT (q < x) & (q > y) ::
                        SOME(r: type_2 :: p(z,q,r)))
END
```

Figure 4-13. **Spec FUNCTION "with_quantifiers"**

quantifiers, respectively). The CHECK_PKG body generated for "with_quantifiers" is presented in Figure 4-14. The **QUANTIFIER FUNCTIONS** generation is the code running from line *3* to *13*.

A detailed discussion of **QUANTIFIER FUNCTIONS** is presented with the discussion on Spec-to_Ada **EXPRESSION TRANSLATION**, Chapter IV.

6. **RESPONSE TRANSFORMATION**

The purpose of the **RESPONSE TRANSFORMATION** is twofold:

1. To check whether a function's response to a stimulus is correct.
2. To generate an error message when the response is not correct.

Using the MESSAGE, REPLY, and termination condition values (i.e., the formal parameters of CHECK), it checks which preconditions hold and checks if corresponding postconditions hold. When the evaluation of a postcondition fails, it generates a three part error message containing:

1. The test set (message values, reply values, and condition).
2. The precondition that applies.
3. The postcondition causing the failure.

The **RESPONSE TRANSFORMATION** generated for the generic square root starts on line *7*, Figure 4-12, and continues to the end of procedure CHECK.

The **RESPONSE TRANSFORMATION** slot is generated from two subtemplates **RESPONSE CASES TRANSFORMATION** or **RESPONSE SET TRANSFORMATION**. Generation of these templates

```
with GEN_ALL_4;                                                    *1*
with GEN_SOME_5;                                                   *2*
package body CHECK_PKG is
     procedure CHECK(condition: condition_type;
                                    x, y: type_1;
                                    z: type_1) is
          preconditions_satisfied: boolean := true;
          q: type_1;                                               *3*
          function ALL_4 return boolean is                         *4*
               value: boolean := true;

               r: type_2;                                          *5*
               function SOME_5 return boolean is                   *6*
                    value: boolean := false;
               begin
                    foreach([r:type_2],GEN_SOME_5,[assurance],[
                         if (value = false) then                   *7*
                              if (true) then                       *8*
                                   if p(z , q , r) then
                                        value := true;
                                   end if;
                              end if;
                         end if;])
                    return value;                                  *9*
               end SOME_5;
          begin
               foreach([q:type_1],GEN_ALL_4,[assurance],[         *10*
                    if (value = true) then
                         if (((q < x)) and ((q > y))) then         *11*
                              if not SOME_5 then                   *12*
                                   value := false;
                              end if;
                         end if;
                    end if;])
               return value;
          end ALL_4;                                               *13*
     begin
          if not (ALL_4) then                                      *14*
               REPORT.ERROR(condition,x, y, z,
               " NOT ALL(q : type_1 SUCH THAT (q<x)&(q<y) ::
                SOME(r : type_2 :: p(z,q,r)))");
          end if;
     end CHECK;
end CHECK_PKG;
```

Figure 4-14. **CHECK_PKG Body of "with_quantifiers" of Figure 4-13**

is predicated on the fact that Spec preconditions and postconditions are assertions; consequently, they can be viewed as functions returning boolean values.

*a. **RESPONSE CASES TRANSFORMATION***

The format of the **RESPONSE CASES TRANSFORMATION** is provided in Figure 4-15. It is a sequence of zero to "m+1" Ada "if statements." The first "m" statements determine if preconditions ("WHENs") are satisfied (line *1*, Figure 4-15). If so, they test the corresponding postconditions ("WHEREs") nested inside and generate an error

```
if (precondition 1) then                    --WHEN precondition 1     *1*
     if not (postcondition 1) then          --WHERE postcondition 1   *2*
          --report error
     end if;
     preconditions_satisfied := true;                                 *3*
end if;
                    .
                    .
                    .
if (precondition m) then                    --WHEN precondition m
     if not (postcondition m) then          --WHERE postcondition m
          --report error
     end if;
     preconditions_satisfied := true;
end if;

if not (preconditions_satisfied) then       --OTHERWISE               *4*
     if not (postcondition (m+1)) then      --OTHERWISE postcondition
          --report error
     end if;
end if;
```

Figure 4-15. **RESPONSE CASES TRANSFORMATION**

message when they fail (line *2*). They also set the flag "preconditions_satisfied" to "true," which simply indicates that "some" precondition has been satisfied (line *3*). Since the "if statements" run in sequence, all preconditions ("WHENs") are tested. This is important because Spec permits overlapping preconditions to exist.

When overlaps exist, it is possible for a message to satisfy more than one precondition. When a message satisfies more than one precondition, the reply must satisfy all of the postconditions associated with those preconditions. [Ref. 1:p. 3-9] For example, if a message satisfies "precondition 1" and "precondition (m-1)," then the REPLY must satisfy both "postcondition m" and "postcondition (m-1)."

The last "if statement," the $(m+1)^{st}$ statement (line *4*) is associated with the Spec "OTHERWISE." It checks the "preconditions_satisfied" variable to determine if any preconditions have been satisfied. If no preconditions have been satisfied, then the final postcondition is checked. The final postcondition must be satisfied if and only if no preconditions are satisfied. For example, the response to a stimulus failing to satisfy preconditions 1 through m, must satisfy "postcondition (m+1)." [Ref. 1:p. 3-9]

The first m if statements are generated right recursively, in association with the "response_cases : WHEN ..." production rule (Template *1*, Figure 4-16). The last if statement is generated in association with the "response_cases : OTHERWISE ..." production rule (Template *2*, Figure 4-16).

```
if **WHEN EXPRESSION LIST TRANSFORMATION** then                    *1*
    **RESPONSE SET TRANSFORMATION**
    preconditions_satisfied := true;
end if;
**RESPONSE CASES TRANSFORMATION**


if not (preconditions_satisfied) then                              *2*
    **RESPONSE SET TRANSFORMATION**
end if;
```

Figure 4-16. ****RESPONSE CASES TRANSFORMATION** Alternatives**

*b. **RESPONSE SET TRANSFORMATION***

When no preconditions are specified in a Spec, the **RESPONSE TRANSFORMATION** is simply a single postcondition check like the one beginning on line *2*, Figure 4-15, which is generated from the **RESPONSE SET TRANSFORMATION** template. It is also a subtemplate of **RESPONSE CASES TRANSFORMATION** alternatives shown in Figure 4-16. As limited by the current implementation, it is derived from the "response_set : reply" production. It is further limited to replies with formal arguments and postconditions or replies with exceptions, which are generated using the templates presented in Figure 4-17 (Templates *1* and *2*, respectively).

*c. **WHEN EXPRESSION LIST TRANSFORMATION***

When Spec preconditions consist of more than one logical expression separated by commas (,), the antecedent checking for its satisfaction is translated as a sequence of those expressions separated by "

```
if not (**EXPRESSION LIST TRANSFORMATION**) then                    *1*
    REPORT.ERROR(condition,
                 **RESPONSE ACTUAL PARAMETERS**,
                 **WHEN ERROR MESSAGE**, " NOT ",
                 **EXPRESSION TRANSLATION**);
end if;

if not (condition = **EXCEPTION NAME**_condition) then              *2*
    REPORT.ERROR(condition,
                 **RESPONSE ACTUAL PARAMETERS**,
                 " NOT EXCEPTION ",
                 **EXCEPTION NAME**);
end if;
```

Figure 4-17. ****RESPONSE SET TRANSFORMATION** Alternatives**

and then " as shown in line *1*, Figure 4-18. It is referred to as the **WHEN EXPRESSION LIST TRANSFORMATION**. This serves as a short circuit whenever any of the expressions in the precondition fail.

```
--WHEN (expression 1) , (expression 2)
if ((expression 1) and then (expression 2)) then                    *1*
    --WHERE (expression 3) , (expression 4)
    if not (expression 3) then                                      *2*
        "report error"
    end if;

    if not (expression 4) then                                      *3*
        "report error"
    end if;
    postconditions_satisfied := true;
end if;
```

Figure 4-18. **Translation of Precondition and Postcondition with Multiple Expressions**

*d. **WHERE EXPRESSION LIST TRANSFORMATION***

When Spec postconditions consist of more than one logical expression separated by a comma (,), a sequence of "if statements," one for each expression, is generated to check the postcondition. This is shown in lines *2* and *3*, Figure 4-18, and is referred to as the **WHERE EXPRESSION LIST TRANSFORMATION**. This serves to isolate the portion(s) of the postcondition causing the failure. A consequence of this is that more than one error may be generated from a response's failure to satisfy a single postcondition. The generic square root function is an example of a Spec with a multiple expression postcondition. The "if statements" on lines *8* and *9*, Figure 4-12, check the independent expressions of the postcondition.

*e. Spec-to-Ada **EXPRESSION TRANSLATION***

The formats of Figures 4-15 and 4-18 rely on the fact that all Spec preconditions and postconditions have values of type boolean. All Spec expressions are translated into a semantically equivalent Ada form, a summary of which is provided in Appendix G. Three basic translation schemes are used (shown in Figure 4-19).

(1) Translation Scheme 1. This scheme is for arity-2 expressions not containing the Spec "not" symbol (~). The expression maintains its original form. The Ada operator used in the translation is the one considered to provide the closest "match" to the operation of the Spec operator. A premise is that the Ada operator selected is defined for

Translation Scheme 1

expr Spec_op expr e.g., x || f(x)	expr_trans Ada_op expr_trans x & f(x)

Translation Scheme 2

expr ~Spec_op expr e.g., x ~>= f(x)	NOT (expr_trans Ada_op expr_trans) NOT (x >= f(x))

Translation Scheme 3

QUANTIFIER(formals :: e) e.g., ALL(x,y: t SUCH THAT p(x) :: q(x,y,z))	QUANTIFIER_NAME_XX ALL_01
expr <=> expr e.g., a <=> b	iff(expr, expr) iff(a,b)

Figure 4-19. **Expression Translation Schemes**

the operands types. For example, the first example in Figure 4-19 shows that the Spec concatenation operator "||" is translated to the Ada string concatenation operator "&."

(2) Translation Scheme 2. This scheme is used for arity-2 expressions containing the Spec "not" symbol. These expressions are translated like the first scheme, except the expression is enclosed in parentheses and preceded by "NOT." An exception is made for the Spec "~=" operator for which the Ada "/=" operator exists.

(3) Translation Scheme 3. The final scheme is used for all other expressions and for the distinguished expressions for which a unique translation was desired. A discussion of Spec QUANTIFIERS, IF,

and "<=>" is presented in the following Section. Refer to Appendix G for information concerning translations and assumptions of remaining expressions.

f. Distinguished Expression Translations

QUANTIFIERS, IF, and "<=>" expressions were translated uniquely as explained here.

(1) Spec QUANTIFIERs. Spec QUANTIFIER expressions include: ALL, SOME, NUMBER, SUM, PRODUCT, SET, MAXIMUM, MINIMUM, UNION, and INTERSECTION. QUANTIFIERs pose four problems:

1. QUANTIFIERs are not suitable for the straightforward "in-line" translation scheme shown in Figures 4-15 and 4-18.
2. QUANTIFIERs can introduce local variables into the Spec whose scope extends throughout the QUANTIFIER [Ref. 1:p. 103].
3. QUANTIFIERs may be nested inside other QUANTIFIERs.
4. Evaluation of QUANTIFIERs involves cycling through the ranges of all the locally declared variables, before their "values" can be determined.

These problems are solved by implementing QUANTIFIERs as parameterless functions which return the "value" of the QUANTIFIER. The parameterless QUANTIFIER function call is used as the "expression" in the evaluation of preconditions or postconditions (e.g., "ALL_4" on line *14*, Figure 4-14). Local variables are declared with the functions (e.g., "q" and "r" on lines *3* and *5*). Nested quantifiers are implemented as nested functions (e.g., SOME_5, line *5*) and ITERATORs are used to cycle through the ranges of values of the declared variables (e.g., lines *7* and *10*).

The logic required to determine the "value" of the QUANTIFIER is dependent on the type of QUANTIFIER (e.g., ALL, SOME, SUM, etc.). Examination of the **ALL QUANTIFIER FUNCTION**, Figure 4-20, illustrates the common elements of the scheme used to evaluate every QUANTIFIER:

1. Determine whether the "iterated" "values" meet the restrictions (line *6*) and
2. If so, check the postcondition and assign/adjust the function's return "value" as appropriate (line *7*).

```
**QUANTIFIER PARAMETER SPECIFICATIONS**                          *1*
function ALL_**QUANTIFIER LINE NUMBER** return boolean is
    value: boolean:= true;
    **NESTED QUANTIFIER FUNCTIONS**                              *2*
begin
    foreach([**ITERATOR LOOP VARIABLES**],                       *3*
        GEN_ALL_**QUANTIFIER LINE NUMBER**,[assurance],[         *4*
            if (value = true) then                               *5*
                if **QUANTIFIER SUCH THAT TRANSLATION** then     *6*
                    if not **EXPRESSION TRANSLATION** then       *7*
                        value := false;
                    end if;
                end if;
            end if;])
        return value;
end ALL_**QUANTIFIER LINE NUMBER**;
```

Figure 4-20. ****ALL QUANTIFIER FUNCTION** Template**

In the case of the ALL QUANTIFIER, the "ALL" function return "value" is initially "true" (line *2*). Then for each set of iterated values (line *3*):

1. It checks whether the function's return "value" remains "true" (line *5) (by definition if must be true for every case).

2. If so, it checks whether the set of iterated values meets the restrictions (line *6*).

3. If so, it examines the postconditions for contradictions (line *7*).

If a contradiction is found, the function's return "value" gets set to "false." This serves two purposes:

1. It short circuits fu ther examination of iterated values (i.e., ALL is "false," no further checking is required.).

2. It serves as the return "variable" upon completion of the iteration process.

The SOME QUANTIFIER value starts out "false," and searches for a single case that makes it "true" using a slightly modified template.

(2) Spec "IF" Expression. The Spec "IF" expression is unique in that the postcondition ("THEN" consequence) to be checked must be determined from the antecedents of the "IF" and "ELSE IF" parts. The "truth" of the expression is determined by the postcondition associated with the first true antecedent or the final consequence (postcondition). Function "conditional" of Figure 4-21 illustrates the point. The Spec has no precondition [WHEN]. First "x < 0.0" is checked; if this is true then "y = 'less than zero'" is the postcondition to be checked. Otherwise "x = 0.0" is checked; if this is true, then "y = 'equal to zero'" is the postcondition. If neither is satisfied, then "y ='greater than zero'" is the postcondition.

```
FUNCTION conditional
    MESSAGE(x: float)
        REPLY(y: enum_string)
            WHERE IF      x < 0.0   THEN   y = less_than_zero
                ELSE IF   x = 0.0   THEN   y = equal_to_zero
                ELSE                       y = greater_than_zero
              FI
END
```

Figure 4-21. **Spec Function "conditional"**

As illustrated in Figure 4-22, the implementing logic is nearly a direct conversion to the Ada "if" statement except that the post-conditions are converted to "if NOT (postcondition) ..." expressions. The difference between the Spec "IF" and "WHEN" is that the short circuit effect of the "IF" provides no overlap. As currently implemented, the "IF" statement may only be used as a top level operator in a postcondition expression list (i.e., if used, it may not be used as an expression within another expression (including itself) or as part of an precondition expression list).

(3) Spec "<=>" and "=>" Expressions. The "<=>" expression was translated as a function implementing the logical definition of "if and only if." In effect, "x <=> y" means "x => y & y => x." The "=>" was also implemented as a function satisfying the definition: "x => y" means "~x | y." The expression translation used is a function call with logical expressions as actual parameters as shown in Figure 4-19.

```
if ((x < 0.0)) then
     if not ((y = less_than_zero)) then
          REPORT.ERROR(condition, x, y,
               " IF x<0.0 NOT y = less_than_zero");
     end if;
elsif(x = 0.0)then
     if not ((y = equal_to_zero)) then
          REPORT.ERROR(condition, x, y,
               " IF-ELSE_IF x = 0.0 NOT y = equal_to_zero");
     end if;
else
     if not ((y = greater_than_zero)) then
          REPORT.ERROR(condition, x, y,
               " IF-ELSE NOT y = greater_than_zero");
     end if;
end if;
```

Figure 4-22. **"IF" Checking Logic Applied to "conditional" of Figure 4-21**

7. **CONCEPT SUBPROGRAM BODIES**

The purpose of **CONCEPT SUBPROGRAM BODIES** is to implement simple Spec CONCEPTs as functions. The purpose of the slots of Figure 4-23 can be determined from context. The CONCEPT "approximates" (Figure 1-4) is an example. The current implementation requires that the leftmost expression in the concept statements (expression list) be the return parameter and it must be followed by either "=" or "<=>." The leftmost expression (i.e., the return variable) and the "=" or "<=>" are replaced with "return," resulting in the Ada "return" expression (line *13*, Figure 4-12).

```
function **CONCEPT DESIGNATOR**(**CONCEPT PARM SPECS**)
                                        return **TYPE MARK** is
        **CONCEPT DECLARATIVE PART**
begin
        **CONCEPT SEQUENCE OF STATEMENTS**
end **CONCEPT DESIGNATOR**
```

Figure 4-23. ****CONCEPT SUBPROGRAM BODIES** Template**

Generation of the functions is accomplished using the template of Figure 4-23. The **CONCEPT SEQUENCE OF STATEMENTS** is implemented using the expression translations discussed in the section on **RESPONSE TRANSFORMATION** with the following variation. To identify the cases in which a return statement should be generated, two attributes "cr_parm" and "is_leftmost" are defined. The attribute "cr_parm" contains either the name of the concept return parameter or "false." The attribute "is_leftmost" contains "true" or "false," indicating whether the expression is the leftmost expression in the subtree. At the "<=>" and "=" expressions, if cr_parm is not "false" and leftmost is "true," then a "return" statement for producing the value of a concept function is generated; otherwise, the normal expression translation is generated.

E. REPORT TEMPLATE

A pruned version of the REPORT template is presented in Figure 4-24. The complete template may be found in Appendix F. It has four slots to be filled in:

1. **PARAMETER SPECIFICATIONS**

```
with TEXT_IO; use TEXT_IO;
with IMPLEMENTATION; use IMPLEMENTATION;
with CONDITION_TYPE_PKG; use CONDITION_TYPE_PKG;
package REPORT is
     procedure ERROR(condition: condition_type;
                     **PARAMETER SPECIFICATIONS**                    *1*
                          msg: string);
          procedure OPEN;
          procedure WRITE_INSTANCE_HEADER(msg: string);
          procedure INCREMENT_SAMPLES_TESTED;
          procedure WRITE_INSTANCE_STATS;
          procedure CLOSE;
     end REPORT;

package body REPORT is
     procedure OPEN is
     begin
                     ...
          PUT_LINE(outfile,"
                     **FUNCTION DESIGNATOR** Test Results");        *2*
                     ...
     end OPEN;
                                   .
                                   .
                                   .
     procedure ERROR(condition: CONDITION_TYPE;
                     **PARAMETER SPECIFICATIONS**;    *3*
                     msg: string) is
     begin
                     ...
                     **PARAMETER PUT STATEMENTS**       *4*
                     ...
     end ERROR;
                                   .
                                   .
                                   .
end REPORT;
```

Figure 4-24. **REPORT Template**

2. **FUNCTION DESIGNATOR**

3. **PARAMETER SPECIFICATIONS**

4. **PARAMETER PUT STATEMENTS**

The slots are located on lines *1* through *4*, respectively. All of the slots except **PARAMETER PUT STATEMENTS** have been previously discussed.

The purpose of the **PARAMETER PUT STATEMENTS** slot is to output MESSAGE and REPLY parameters and their values. Two "PUT" statements and a "NEWLINE" are generated for each parameter.

The generic square root example of Figure 1-4 has one MESSAGE parameter "x" and one REPLY parameter "y." The **PARAMETER PUT STATEMENTS** generated for that example are presented in Figure 4-25.

The **PARAMETER PUT STATEMENTS** slot is generated analogously to MAIN_PKG **GENERIC OBJECT GETS**.

F. CONDITION_TYPE_PKG TEMPLATE

The CONDITION_TYPE_PKG template is presented in Figure 4-26. It has only one slot: **CONDITION TYPES**. The purpose of **CONDITION_TYPES** is to enumerate a value for each exception contained in the Spec and enumerate the two fixed condition values: "normal" and "unspecified_exception."

The generic square root example of Figure 1-4 had only one exception: "imaginary_square_root." The CONDITION_TYPE_PKG generated for the generic square root example is presented in Figure 4-27.

```
PUT(outfile,"x =");
PUT(outfile,x);
NEW_LINE(outfile);
PUT(outfile,"y =");
PUT(outfile,y);
NEW_LINE(outfile);
```

Figure 4-25. **REPORT **PARAMETER PUT STATEMENTS** for Figure 1-4**

```
with TEXT_IO; use TEXT_IO;
package CONDITION_TYPE_PKG is
    type CONDITION_TYPE is (**CONDITION TYPES**);
    package CONDITION_TYPE_IO is new
                        ENUMERATION_IO(CONDITION_TYPE);
end CONDITION_TYPE_PKG;
```

Figure 4-26. **CONDITION_TYPE_PKG Template**

```
with TEXT_IO; use TEXT_IO;
package CONDITION_TYPE_PKG is
    type CONDITION_TYPE is    (normal, unspecified_exception,
                               imaginary_square_root_condition);
    package CONDITION_TYPE_IO is new
                        ENUMERATION_IO(CONDITION_TYPE);
end CONDITION_TYPE_PKG;
```

Figure 4-27. **CONDITION_TYPE_PKG for "generic square root"**

Generation of **CONDITION TYPES** is straightforward. A single condition types is the concatenation of the Spec exception name and "_condition." An exception may be used more than once in a single Spec; consequently, a map is used to guard against duplication when merging condition types at the apexes of the response and response_cases sub-trees (i.e., response and response_cases production rules.).

G. GENERATOR TEMPLATE

The GENERATOR template is presented in Figure 4-28. It has only one slot: **GENERATOR LOOP VARIABLES**.

```
include(/n/suns2/work/student/depasqua/MACROS/generator.m4)
--**put with and use clauses here**--
generator(GENERATOR, [assurance: float],
        [**GENERATOR LOOP VARIABLES**],
 [is
        --**put required declarations here**--
begin
        --**put statements to generate values here**--
        generate(--**put generated values here**-);
        --**put more statements here as required**--
end GENERATOR;])
```

Figure 4-28. **GENERATOR Template**

The purpose of **GENERATOR LOOP VARIABLES** is to specify the MESSAGE parameters and their corresponding types.

The generic square root example of Figure 1-4 had only one MESSAGE parameter "x" of type "float." The GENERATOR generated for the generic square root example is presented in Figure 4-29. When more

than one MESSAGE parameter exists, the parameters and their types are specified in the order they appear in the Spec.

```
include(/n/suns2/work/student/depasqua/MACROS/generator.m4)
--**put with and use clauses here**--
generator(GENERATOR, [assurance: float], [x:float],
[is
        --**put required declarations here**--
begin
        --**put statements to generate values here**--
        generate(--**put generated values here**-);
        --**put more statements here as required**--
end GENERATOR;])
```

Figure 4-29. **GENERATOR for "generic square root"**

H. ITERATORS TEMPLATE

The ITERATORS template is presented in Figure 4-30. It is identical to the GENERATOR template except:

1. Each ITERATOR is assigned a unique name "GEN_**FUNCTION_DESIGNATOR**" instead of the standard name "GENERATOR."
2. The **GENERATOR LOOP VARIABLES** correspond to local declarations of the QUANTIFIER expression instead of the MESSAGE parameters.
3. If more than one iterator is generated (i.e., whenever the Spec contains more than one QUANTIFIER), all iterators are concatenated into a single file.

The ITERATORS generated for the "with_quantifiers" function of Figure 4-13 is presented in Figure 4-31. Two ITERATORS were generated and concatenated corresponding to the SOME and ALL quantifiers

```
include(/n/suns2/work/student/depasqua/MACROS/generator.m4)
--**put with and use clauses here**--
generator(**GEN_FUNCTION_DESIGNATOR**, [assurance: float],
        [**GENERATOR LOOP VARIABLES**],
[is
        --**put required declarations here**--
begin
        --**put statements to generate values here**--
        generate(--**put generated values here**-);
        --**put more statements here as required**--
end **GEN_FUNCTION_DESIGNATOR**;])
```

Figure 4-30. **ITERATORS Template**

```
include(/n/suns2/work/student/depasqua/MACROS/generator.m4)
--**put with and use statements here**--
generator(GEN_SOME_5,[assurance: float],[r:type_2],                    *1*
[is
        --**put any required declarations here**--
begin
        --**put iterating statements here**--
        generate(--**put generated values here**);
        --**put more statements here as required**--
end GEN_SOME_5;])                                                      *2*

--**put with and use statements here**--
generator(GEN_ALL_4,[assurance: float],[q:type_1],                     *3*
[is
        --**put any required declarations here**--
begin
        --**put iterating statements here**--
        generate(--**put generated values here**);
        --**put more statements here as required**--
end GEN_ALL_4;])                                                       *4*
```

Figure 4-31. **ITERATORS for "with_quantifiers"**

contained in the Spec. The names "GEN_SOME_5" and "GEN_ALL_4" are the ITERATOR names, which are the concatenation of "GEN_" and the "function designators" enumerated in procedure CHECK to evaluate the QUANTIFIERS (lines *1*, *2*, *3*, and *4*). Finally, the specifications "i:type_2" and "q:type_1" correspond to the local variables of the QUANTIFIERS (lines *1* and *3*).

V. EXTENSIONS

The Module Driver and Output Analyzer Generator's greatest value is as a prototype to demonstrate the viability of automatically generating testing code, not as a system for production use. It will produce working Module Driver and Output Analyzers for only the simplest of Spec functions. This chapter outlines some suggestions for extending and improving the system or using alternate methodologies.

A. GENERIC TYPES

The MDOAG of this research generates test code for generic object parameters but not generic type parameters (i.e., type "type"), union types, or universal types (i.e., type "any"). Figure 5-1 is an example of a function with a generic type. Originally, work was started to support generic types, but after some effort the work was stopped in favor of concentrating on a smaller Spec subset. This section is devoted to presenting two general approaches that can be used to test Specs containing generic type parameters. The first approach is to generate a "generic" Module Driver and Output Analyzer from a generic Spec. The second approach is to generate a set of Spec instances S_{TS} from the generic Spec and then to generate a set of Module Driver and Output Analyzers M_{TS} from S_{TS}.

1. Generic Module Driver and Output Analyzer

Generation of "Generic" MDOAs from Specs containing generic types can be handled with essentially the same methodology used to

```
FUNCTION max(t: type)
    MESSAGE (q r: t)
        REPLY(s: t) IF q >= r THEN s = q ELSE s = r FI
END
```

Figure 5-1. **Spec "max" Using Generic Type**

implement generic value parameters in the current prototype. There are, however, several distinctions. The first distinction is that the MDOAG prototype generates what is essentially a generic MDOA wherein the instantiation occurs after the values are read from the external file "test_parameters." However, no Ada construct exists which allows type parameters to be read in from a file and used to instantiate formal-generic type parameters. Consequently, the actual types of the Spec must be provided to the MDOAG in advance. A proposal for introducing the types is presented later.

The second distinction between generic object parameters and generic type parameters, as they apply to the MDOAG, is that "types" carry a great deal of important baggage in the form of operations. In a generic MDOA, just as the generic value parameters were passed from module to module (e.g., generic parameter "precision" passed by instantiation in the generic square root example), the types *and* their operations must be passed from module to module. At first consideration, this seems trivial:

1. Generate a "limited private" generic type parameter declaration for each generic type in the Spec.

2. Generate a subprogram parameter (default box "<>")[1] for each of the operations of the type in the Spec.

3. Place a copy of the generic type declarations and subprogram declarations in the generic parts of all the modules.

For example, as proposed above, the generic part of MAIN_PKG would appear as in Figure 5-2. Unfortunately, this is an oversimplification.

```
generic
    type t is limited private;
    with function ">=" (x, y: t) return boolean is <>;
    with function "=" (x, y: t) return boolean is <>;
package MAIN_PACKAGE is ...
```

Figure 5-2. **Generic Part of Package MAIN_PKG for Spec "max"**

a. Generic-Type Declaration

Sole use of the "limited private" generic type declaration is inappropriate. Case-by-case consideration must be made when selecting the "most appropriate" generic type declaration from those available: limited private, private, and "predefined." Predefined generic type declarations are listed in Figure 5-3 [Ref. 2:pp. 249-250].

(1) Generic "limited private" Type Declaration. The general applicability of the generic "limited private" type declaration would seem

[1]Default <> allows for the omission of the actual parameter in the instantiation of modules using the type.

DECLARATION FORMAT	TYPE
type "link" is access "object";	ACCESS
type "enumeration" is (<>);	ENUMERATION
type "integer type" is range <>;	INTEGER
type "fixed element" is delta <>;	FIXED
type "float" is digits <>;	FLOAT
type "constrained" is (array ("index") of element);	CONSTRAINED ARRAY
type "unconstrained" is array ("index" range <>);	UNCONSTRAINED ARRAY

Figure 5-3. **Predefined Generic Type Declarations**

to make it ideal because every type can be represented by this one declaration. However, there are five reasons why this declaration is neither ideal nor sufficient:

1. The "limited private" class contains no implicit "assignment" operator or "test for equality." It is difficult to conceive of many tests that can be conducted without the use of an "assignment" operator to be used for saving variables for later evaluation (e.g., in the MDOA, the assignment statement is used to set the REPLY variable equal to the return value of the function). Consequently, it seems reasonable to require that all types (contained in functions to be tested) have an "assignment" operator defined for them for test purposes (if for no other reason). Ironically, this requirement turns the "limited private" type into a "quasi private" type.

2. The use of "limited private" may unnecessarily restrict developers from using the inherent attributes of "predefined" types. Importation of a predefined type as a generic "limited private" formal parameter essentially strips the type of its attributes. For instance, type "delta <>" imported as "limited private" type loses its useful attributes (e.g., FIRST, LAST, DELTA, etc.). Consequently, a large set of tools is lost to developers. This is particularly inappropriate when a less-general generic type is wholly suitable to meet the spe-

cification. Further, it is envisioned that extensive use of attributes will be employed for iteration and looping constructs in the ITERATOR and GENERATOR modules.

3. The use of "limited private" may cause MDOA-to-function interface incompatibilities. The use of "limited private" type parameter will cause an MDOA-to-function interface incompatibility when an developer uses a less-general, generic type parameter (but sufficient to implement the generic Spec correctly) and the MDOA imports the type unconditionally as "limited private." The interface error will occur when the MDOA attempts to instantiate the generic function [Ref. 2:p. 12-11].

4. The use of limited private type declaration may unnecessarily require the user to manually provide subprocedure specifications. The limited private type declaration requires that the user enumerate every operation as a subprocedure parameter. Consequently, the user will have to provide the information. Generic predefined type declarations have no such requirement.

5. The use of limited private may unnecessarily clutter the MDOA code. Because every operation of the limited private declaration must be explicitly enumerated, the limited private declaration may unnecessarily clutter the code. One generic subprogram parameter will be generated for each operator of each generic type in the generic part of every using module. It is clear that the number of generic procedure parameters will very quickly become unwieldy as the number of generic types and their operators increase.

One reason the use of the generic limited private declaration is required is to test cases when the actual type is implemented as a "limited private" type. This is required to observe the following rule: "If the formal type is not limited, the actual type must not be a limited type." [Ref. 2:p. 12-11] For instance, if the generic MDOA generated from the Spec of Figure 5-1 is to be tested for several implemented types, one of which is type "Lim_Priv" where Lim_Priv was implemented as a limited private type, then the only generic type declaration that may be used to to import it is limited private. Such design decisions are recorded using the Spec pragma "limited" and are potentially available to the MDOAG.

(2) Generic "private" Type Declaration. The generic "private" type parameter is analogous to the limited private in every way except one: it has a built in assignment operator and "test for equality." It should be used to test cases when the actual type "t" is implemented as a private type.

(3) Generic "predefined" Type Declarations. The generic "predefined" type declaration has two advantages over limited private and private declarations.

1. Predefined types implicitly import their operations with them. The need for the user to provide subprocedure parameters is eliminated along with the potential clutter resulting from their generation.
2. The type retains all the attributes associated with the "predefined" type. These attributes may be used freely by developers.

(4) Summary of Generic Type Declarations. Figure 5-4 provides a summary of declaration types. Essentially, the most appropriate generic type declaration is the one that matches the actual type required by the Spec. Error-free instantiation of the implementation will depend upon the generic type declaration used in the implementation. The figure shows the information requirements necessary to generate the proper declarative part.

b. Generic Subprogram Specifications

Generation of the Generic Subprogram Specifications to represent the operations of the types cannot be accomplished by "simply" pulling the operators from the Spec and generating the appropriate specifications. The problem is that the Spec does not contain all of the type

Use When Actual Requirements	Declaration	Actual Type is
limited private	limited private	assignment operator test for equality operator specifications I/O (PUT and GET)
private	private	operator specifications I/O (PUT and GET)
predefined		
access "object"	access	I/O (PUT and GET)
(<>)	enumeration integer	I/O (PUT and GET)
range <>	integer	I/O (PUT and GET)
delta <>	fixed point	I/O (PUT and GET)
digits <>	floating point	I/O (PUT and GET)
array ("index") of "element"	constrained array	I/O (PUT and GET)
array ("index" range <>) of "element"	unconstrained array	I/O (PUT and GET)

Figure 5-4. **Summary of Generic Type Declarations**

operations required by the modules of the generic MDOA. All of the operations of the generic type which are available to the generic modules of the MDOA come from instantiation. MAIN_PKG gets the actual type and its operations from MAIN through instantiation, DRIVER receives them from MAIN_PKG, the generic "function" gets them from DRIVER, etc. Since a Spec abstractly defines the desired behavior of the module, not how to implement it, it will normally specify the behavior with less

operators than it will take to implement and test that behavior. Consider the generic square root example: only six operators were required to specify the desired behavior (i.e., ">=," ">," "*," "abs," "<=," and "-"). However, to implement and test that behavior, four additional operators were required (i.e., ":=," "<," "=," and "+"—see Figures 4-6, 4-12, and Section B of the Sample in Appendix C). Although this was a concrete example, the principle holds in general. If it did not, there would be no advantage in writing a specification. It would be as complex as the implementation. Consequently, all of the types and their operators (not just those in the Spec) must be introduced to the MDOAG to ensure that the operations are available for the implementation of the generator, for the iterators, and for the instantiation of the function (which is assumed to be developed independently). Hence, the operator specifications of Figure 5-4 must include all operators, not just those in the Spec.

c. Spec Trailer

Detailed concrete type information must to available to MDOAG at run time to generate the generic MDOA. Pragmas are one way of importing the information. However, the sheer bulk of type information required precludes inserting a pragma in the test of the Spec. Instead, it is better to append the additional information at the end of the Spec in the form of a Spec trailer. The language Spec, combined with these few additional production rules, forms the MDOAG test language.

The function production rule would be modified by appending the test_data non-terminal, as shown at the top of Figure 5-5. The

```
function:
        optionally_virtual FUNCTION ... END test_data

test_data:
        GENERIC_TYPES_DATA generic_types_data
      |

generic_types_data:
        generic_types_data generic_type_data
      |

generic_type_data:
        "[" type_name BIND generic_op_class BIND op_specs ']'

type_name:
        NAME

generic_op_class:
        LIM_PRIVATE
      | PRIVATE
      | predefined

predefined:
        ACCESS
      | ENUMERATION
      | INTEGER
      | FIXED
      | FLOAT
      | CONSTRAINED_ARRAY
      | UNCONSTRAINED_ARRAY

op_specs:
        op_specs op_spec
      | op_spec

op_spec:
        ADA_subprogram_spec ';'

ADA_subprogram_spec:
(       remaining rules required to ensure proper Ada spec)
```

Figure 5-5. **Spec Trailer Production Rules**

test_data production rules and subrules are also shown in the figure. Using those production rules, the user could augment a Spec to provide the necessary type information without cluttering the Spec. The "Spec" of Figure 5-6 is the "max" sample supplying the required type information assuming the actual type is "limited private." For limited private, Figure 5-4 indicates that an assignment operator, test for equality, all operators, and I/O operations are required. Figure 5-7 shows the Spec when the type is an integer type. Shorthand notation could be developed to reduce the size of the trailer. Attributes and semantic functions can be developed to generate the appropriate generic formal parts for the components of the MDOAG.

```
FUNCTION max{t: type}
        MESSAGE (q r: t)
        REPLY(s: t) IF q >= r THEN s = q ELSE s = r FI END
GENERIC_TYPES_DATA

[t :: LIM_PRIVATE ::    procedure assign(x: in out t; y: t);
                        function "="(x,y:t) return boolean;
                        procedure GET(file: in FILE_TYPE; x: out t);
                        procedure PUT(file: in FILE_TYPE; x:t);
                        function "<"(x,y:t) return boolean;
                        function "<="(x,y:t) return boolean;
                        function ">"(x,y:t) return boolean;

                        (all other LIM_PRIVATE operations)

                        function ">="(x,y:t) return boolean;]
```

Figure 5-6. **Spec "max" Generic "limited private" Type**

```
FUNCTION max[t: type]
        MESSAGE (q r: t)
        REPLY(s: t) IF q >= r THEN s = q ELSE s = r FI
END
GENERIC_TYPES_DATA

[t :: INTEGER ::    procedure GET(file: in FILE_TYPE; x: out t);
                    procedure PUT(file: in FILE_TYPE; x:t);]
```

Figure 5-7. **Spec "max" Generic "predefined" Integer Type**

d. Modifying the Templates

The templates of the MDOA components need to be modified as outlined below in order to generate generic MDOAs from Specs with generic types. Whenever conditional generation of new slots is mentioned below, the condition is based on the existence of generic types in the Spec and the information required to fill the slot is derived from the Spec trailer. Attributes and semantic functions must be created and cause generation in the the normal way.

(1) MAIN. The MDOAG will generate a generic MDOA which will require instantiation by the user. The user will instantiate the MDOA as shown in Figure 5-8. The user provides the new name of the procedure and supplies the type name. It is assumed that the type is implemented in package IMPLEMENTATION.

To support this the MAIN template must be modified as shown in Figure 5-9. A **GENERIC FORMAL PART** slot must be conditionally generated. Also a **MAIN_PKG INSTANTIATION** slot must

```
with IMPLEMENTATION;
procedure **NAME** is new MAIN(IMPLEMENTATION.**TYPE**);
```

Figure 5-8. **Instantiating the Generic MDOA**

```
with REPORT;
with MAIN_PKG;
**GENERIC FORMAL PART**  *1*
procedure MAIN is
   package NEW_MAIN_PKG is new MAIN_PKG(**GENERIC TYPES**);  *2*
begin
   REPORT.OPEN;
   while not (NEW_MAIN_PKG.TESTS_COMPLETE) loop              *3*
      NEW_MAIN_PKG.GET_TEST_PARAMETERS;                      *4*
      NEW_MAIN_PKG.EXECUTE_TEST;                             *5*
   end loop;
   REPORT.CLOSE;
end MAIN;
```

Figure 5-9. **Modified MAIN Template**

be conditionally generated (line *2* is an example of a generated slot). The instantiation statement will rename MAIN_PKG (line *2*); therefore, requests for services must reflect the new name (lines *3* through *5*).

(2) MAIN_PKG. A conditionally generated **GENERIC FORMAL PART** slot must be added to the MAIN_PKG template. It will be identical to MAIN's. The **DRIVER INSTANTIATION OR RENAMING DECLARATION** slot will require modification to provide the actual type parameters for the instantiation. There is no need to include the

operators in the instantiation statement. The default <> will ensure appropriate operator instantiation; however, care must be taken to ensure a standard ordering of the type and value parameters is maintained. A consistent strategy is to delineate the type parameters first, followed by the value parameters.

(3) DRIVER. The **GENERIC FORMAL PART** slot must be modified to conditionally include the declarations associated with the generic types. The order of the generic parameter declarations must be consistent with MAIN_PKG. The **INSTANTIATIONS OR RENAMING DECLARATIONS** slot must conditionally include the actual types. The foreach macro used in DRIVER must be modified to accept **GENERIC TYPES** so that, upon expansion, it appropriately instantiates the GENERATOR.

(4) CHECK_PKG. The **GENERIC OBJECT DECLARATIONS** slot must conditionally include type information. Consistent ordering of the parameters must be maintained. A **REPORT ERROR INSTANTIATION** slot must be added to the declarative part of procedure CHECK.

(5) REPORT. A **GENERIC PART** slot must be added to procedure ERROR to conditionally generate a generic part. The I/O operations used in procedure ERROR must come through instantiation for limited private and private types. To be consistent, if there are generic type parameters, then procedure ERROR is generic.

(6) GENERATOR and ITERATORS. An alternate GENERATOR and ITERATORS macro shell must be generated when generic types

are in the Spec. The alternate template need only be named differently and contain a **GENERIC PART** slot. An additional macro must be written for this alternate template which properly positions the **GENERIC PART** in the generic part of the GENERATOR procedure.

2. Spec Instantiation

An alternate approach to testing for conformance to a generic Spec is to implement a "Spec instantiater." The purpose of the instantiater would be to generate a Spec instance from a generic Spec. Using the instantiater, the user would generate a "test set" of Spec instances, one for each of the combinations of generic parameters to be tested. Then, using the MDOAG, the user would generate a "MDOA instance" for each Spec instance in the test set. Finally, the user would execute each MDOA instance, independently generating a report for that instance.

Under the Spec instantiater method, instantiation of the implemented function is conducted by the user. The MDOAG generates code for that instantiated function. No generic formal parts or instantiation statements are generated for any of the components. The MDOAG generates a "with IMPLEMENTATION" statement for each MDOA component requiring access to the type(s), the assumption being that the type(s) is(are) implemented in the IMPLEMENTATION package.

The demonstrated "assurance" desired from a test is entered as a Spec trailer or pragma, so that no "test_parameters" file is required of the user.

Figures 5-10 and 5-11 demonstrate how the Spec instantiation method works. It is desired to test Spec "max" of Figure 5-1 for types

"lim_priv" and "integer," where lim_priv is some user defined limited private type and integer is the standard Ada integer type. Further, the user desires assurances of 0.001 and 0.003, respectively. The user would augment the Spec "max" with the "instance generation trailer," as shown in Figure 5-10. The trailer is similar to a Spec instantiation statement bounded to the assurance desired for that instantiation. The augmented Spec would be submitted to the Spec instantiater, producing a file containing two instantiated Specs as shown in Figure 5-11. The user would then instantiate the Ada function "max" as implemented for type lim_priv. Then the lim_priv instantiation would be submitted to the MDOAG, which would generate a MDOA instance for type lim_priv. (The assurance would be hard-coded into the generator and iterator shells.) The user would complete the generator and iterators as before and compile the MDOA. The user would execute the MDOA and view the results. The process would be repeated for type integer.

```
FUNCTION max(t: type)
        MESSAGE(q r: t)
        REPLY(s: t) IF q >= r THEN s = q ELSE s = r FI
END

GENERATE_INSTANCES
[max(lim_priv)::ASSURANCE(0.001)      max(integer)::ASSURANCE(0.003)]
```

Figure 5-10. **Spec "max" With Instance Generation Trailer**

```
FUNCTION max(t: lim_priv)
        MESSAGE(q r: lim_priv)
        REPLY(s: lim_priv) IF q >= r THEN s = q ELSE s = r FI
END
TEST_ASSURANCE(0.001)

FUNCTION max(t: integer)
        MESSAGE(q r: integer)
        REPLY(s: integer) IF q >= r THEN s = q ELSE s = r FI
END
TEST_ASSURANCE(0.003)
```

Figure 5-11. **Results of Instantiation of Figure 5-12**

It would be a simple matter to create the instance generator using the Kodiyak application generator. The trailer productions would have to be developed, which is also trivial. Finally, the MDOAG would actually be much simpler than even the current prototype (assuming the same capability).

3. Recommendation for Implementation

Two methods were presented for "extending" the MDOAG to accommodate generic types. Of the two methods presented, the Generic Instantiation method is the simplest to implement and is the recommended approach. The generic MDOAG approach is more intuitive but requires more overhead and imposes more requirements on the user. It was the approach originally conceived, upon which much of the structure of the prototype is based.

B. MULTIPLE MESSAGE MODULES

The MDOAG prototype generates MDOAs capable of testing single-service functions. Spec functions normally provide a single service, but they may be specified to provide more than one service. A Spec function providing more than a single service is represented in the usual way, except that there is more than one stimulus (MESSAGE) with associated responses. Each service provided is distinguished by its formal message name [Ref. 1:p. 3-9]. This section provides a description of how to extend the current system to generate MDOAs capable of testing multiple-service functions.

The testing scheme proposed is to test each service independently and in series. To accomplish this, most of the MDOA components and the MDOAG code require relatively small modifications as follows:

1. MAIN and MAIN_PKG

MAIN and MAIN_PKG require no changes.

2. DRIVER

The multiple-service DRIVER template and its subtemplate are presented in Figures 5-12 and 5-13. The multiple-service DRIVER template is identical to the single-service DRIVER presented in Figure 4-6, except that:

1. The "with GENERATOR;" statement has been exchanged for a **GENERATOR WITH CLAUSES** slot.
2. The declarations and the body of procedure DRIVER have been removed and consolidated in a single subtemplate called **SERVICE DRIVER**. **SERVICE DRIVERS** (plural) is the concatenation of one or more **SERVICE DRIVER** slots.

The multiple services DRIVER is described below.

```
include(/n/suns2/work/student/depasqua/MACROS/generator.m4)
**GENERIC FORMAL PART**
procedure DRIVER(assurance:  in FLOAT);

**GENERATOR WITH CLAUSES**
with CHECK_PKG;
with REPORT; use REPORT;
with IMPLEMENTATION; use IMPLEMENTATION;
with CONDITION_TYPE_PKG; USE CONDITION_TYPE_PKG;

procedure DRIVER is (assurance: in float) is
begin
     **SERVICE DRIVERS**
end DRIVER;
```

Figure 5-12. **Multiple Services DRIVER Template**

```
--DRIVE **MESSAGE FORMAL NAME**
declare
     condition: condition_type := normal;
     **PARAMETER SPECIFICATIONS**
     **INSTANTIATIONS OR RENAMING DECLARATIONS**
begin
     REPORT.WRITE_INSTANCE_HEADER;
     foreach[[(**GENERATOR LOOP VARIABLES**),**GENERATOR**,
               [(assurance)], [
          begin
               **FUNCTION CALL**
               condition := normal;
          exception
               **EXCEPTION_WHEN_CLAUSES**
               when others =>
                    condition := unspecified_exception;
          end;
          BLACK_BOX.CHECK(condition,
                                  **FORMAL MESSAGE ACTUAL PARMS**);
          INCREMENT_SAMPLES_TESTED;]]
     REPORT.WRITE_INSTANCE_STATS;
end;
```

Figure 5-13. ****SERVICE DRIVER** Template**

*a. **GENERATOR WITH CLAUSES***

The purpose of the **GENERATOR WITH CLAUSES** slot is to generate a "with" clause for the distinct generators which provide test input values for the different services. Each service requires its own generator. The generator name for each service is "GEN_**MESSAGE FORMAL NAME**," where **MESSAGE FORMAL NAME** is the formal message name taken verbatim from the Spec. Generating **GENERATOR WITH CLAUSES** is similar to generating **QUANTIFIER WITH CLAUSES** presented in Chapter IV.

*b. **SERVICE DRIVERS***

The purpose of the **SERVICE DRIVERS** slot is to generate a sequence of DRIVERS providing the same functionality that procedure DRIVER did when driving single-service modules. **SERVICE DRIVER** is essentially the procedure DRIVER presented in Figure 4-6, with the following exceptions:

1. **SERVICE DRIVER** is an Ada block instead of a procedure.
2. The comment "--DRIVE **MESSAGE FORMAL NAME**" has been added at the top of the block to assist the user in associating the **SERVICE DRIVER** slots to the service.
3. "GENERATOR" in the "foreach" macro has been changed to the **GENERATOR** slot which is the name of the generator procedure that provides test input values to the particular service. It is the same "GEN_**MESSAGE FORMAL NAME**" used in **GENERATOR WITH CLAUSES**.

There are many alternative methods which could have been chosen to extend DRIVER to accommodate multiple service modules. The key reason for selecting this method is to ensure that there are no naming conflicts in the DRIVER component. Actual parameters and formal

parameters declared in a Spec service are local to that service; consequently, different services may use the same names [Ref.1:p. 3-103]. Had a "non-block" style implementation been proposed, steps would have to be taken to ensure that identifiers representing message and response parameters are not multiply defined. Using a block style keeps the identifiers "alive" only as long as they are used; hence, no renaming or checking is required.

If individual procedures are preferred over Ada block statements, then the block statements may be generated as procedures instead (using "**FORMAL MESSAGE NAME**_DRIVER" as the procedure name). To avoid name conflicts due to overloaded message names, **FORMAL MESSAGE NAME** should include distinguishing postfixes for overloaded operators, such as "_1," "_2," etc. These procedures must be placed in the DRIVER declarations section and **SERVICE DRIVERS** must be replaced by the sequence of calls to those procedures.

3. CHECK_PKG

The modifications required to CHECK_PKG are very similar to those described for DRIVER. Figures 5-14 and 5-15 present the multiple-service check package and its subtemplates pertinent to this discussion. The single CHECK procedure specification of Figure 4-11 has been replaced by the **CHECK PROCEDURE SPECIFICATIONS** slot and the CHECK procedure body has been replaced by the **CHECK SUBPROCEDURES BODIES** slot.

```
include(/n/suns2/work/depasqua/MACROS/generator.m4)
with REPORT; use REPORT; with MDOAG_LIB; use MDOAG_LIB;
with IMPLEMENTATION; use IMPLEMENTATION;
with CONDITION_TYPE_PKG; use CONDITION_TYPE_PKG;
generic
     assurance: float;
     **GENERIC OBJECT DECLARATIONS**
package CHECK_PKG is
     **CHECK PROCEDURE SPECIFICATIONS**
end CHECK_PKG;

**QUANTIFIER WITH CLAUSES**
package body CHECK_PKG is

     **CONCEPT SUBPROGRAM SPECIFICATIONS**

     **CHECK PROCEDURE BODIES**

     **CONCEPT SUBPROGRAM BODIES**
end CHECK_PKG;
```

Figure 5-14. Multiple Service CHECK_PKG Template

```
                **CHECK PROCEDURE SPECIFICATIONS**

     procedure CHECK_**MESSAGE FORMAL NAME**(
                                condition: condition_type;
                                **PARAMETER SPECIFICATIONS**):

                **CHECK PROCEDURE BODIES**

     procedure CHECK_**MESSAGE FORMAL NAME**(
                                condition: condition_type;
                                **PARAMETER SPECIFICATIONS**) is
          preconditions_satisfied: boolean := false;
          **QUANTIFIER FUNCTIONS**
     begin
          **RESPONSE TRANSFORMATION**
     end CHECK_**MESSAGE_FORMAL_NAME**;
```

Figure 5-15. ****CHECK PROCEDURE SPECIFICATIONS** and **CHECK PROCEDURE BODIES** Subtemplates**

a. *****CHECK PROCEDURES SPECIFICATIONS*****

The purpose of the **CHECK PROCEDURES SPECIFICATIONS** slot is to declare a sequence of CHECK procedures, one for each service in the module. Individual procedure specifications are identical to the single CHECK specification of the single-service CHECK except for the procedure name. The procedure names have been augmented to include the formal message name as shown in Figure 5-15.

b. *****CHECK PROCEDURES BODIES*****

The purpose of the **CHECK PROCEDURES BODIES** slot is to generate the code required to check the services of multiple service modules. Individual procedures are the same as the single CHECK procedure of Figure 4-11 except that they reflect the name of the individual service they check, as shown in Figure 5-15.

c. *****RESPONSE TRANSFORMATION*****

The **RESPONSE TRANSFORMATION** slot of Figure 5-15 must also be changed. The "REPORT.ERROR" procedure call must also reflect the name of the service being checked (i.e., "REPORT.ERROR_**FORMAL MESSAGE_NAME**").

4. REPORT

The REPORT template must provide an individual ERROR procedure for each service. This is accomplished by replacing the single ERROR specification and body with **ERROR SPECIFICATIONS** and **ERROR BODIES**, respectively. The changes required are analogous to the changes in CHECK_PKG.

5. CONDITION_TYPE_PKG

The CONDITION_TYPE_PACKAGE must be changed to include all exceptions enumerated in the Spec. Essentially, condition types from each service must be merged prior to placing them in the template. In addition, a mechanism must be provided to ensure that duplicate exceptions are not enumerated in the slot. Kodiyak provides no handy solution for this problem. A means to accomplish this is to declare a condition type map (e.g., condition_entered(condition_type -> boolean) where condition_type and boolean are "strings") which is initialized almost everywhere "false." This map and the attribute representing the **CONDITION TYPES** should be passed into each message subtree. Prior to loading each condition type, check for its existence in the map. If it is not in the map, concatenate the condition type to **CONDITION TYPES** and add it to the map; otherwise, do not. Although the logic is simple, much more code is required to make this change than for any of the other changes required to extend the MDOAG to handle multiple-service messages.

6. GENERATOR

The GENERATOR template must be changed so that the name of the generator procedure reflects the service for which it provides input values (i.e., each place where "GENERATOR" appears in Figure 4-28 must be to **GENERATOR NAME**, where **GENERATOR NAME** is "GEN_**FORMAL MESSAGE NAME** as previously described). All the GENERATORS generated for the service should be concatenated into the single file "input_generators.m4," as ITERATOR(S) are currently done.

7. ITERATORS

The ITERATORS template requires no changes, but the ITERATORS generated by each service must be concatenated onto the iterators.m4 file.

8. MDOAG Code

The MDOAG code (Appendix A) must be changed to reflect the changes in the templates discussed above. In addition, some movement of the templates or portions of the templates in the parse tree is necessary. For instance, **SERVICE DRIVER** should be located at the top of the message subtree rather than at the top of the function subtree. The reasons for the change are:

1. All of the data required to complete the template are present within the message subtree.
2. Concatenation of multiple **SERVICE DRIVER** slots into a single **SERVICE DRIVERS** slot is best handled at the top of the messages subtree.

Appropriate attributes must be declared to handle the passage of information for the new templates.

Analogous changes to the code are required for generating multiple GENERATORS, **CHECK PROCEDURE SPECIFICATIONS**, and **CHECK PROCEDURE BODIES**.

VI. CONCLUSIONS

A. FEASIBILITY

Implementation of a Module Driver and Output Analyzer Generator (MDOAG) for Spec Functions using the Kodiyak Applications Generator is feasible. An operational MDOAG has been implemented for a small subset of Spec FUNCTIONs.

Assertions expressed in Spec can be converted into Ada code, which checks adherence to those assertions. The key aspect is that each Spec expression must be converted into an Ada expression which "returns" the boolean value of the evaluated expression. For most Spec expressions, translations were almost verbatim. For others, the expressions had to be translated into Ada functions which returned the value of the evaluated expression.

B. TEMPLATE METHODOLOGY

The template method developed in Reference 15 proved to be a very good way to generate the code. It was found that large templates can be broken down into subtemplates. The subtemplates serve to modularize the attribute grammar code into functional units which are easily understood and changed when required.

It was also found that readability of attribute grammar code is enhanced when code generation is based on an attribute with a meaningful name, whose value may be determined by the outcome of some

complicated logic, rather than generating code directly from the complex conditional logic and the data directly available in the source code.

C. KODIYAK USER INTERFACE

Kodiyak can be greatly improved with a better user interface. Kodiyak is criticized because it has only one high-level data type (i.e., map) for which there are very few operations [Ref. 15:p. 81; Ref. 16:p. 70]. This research concurs. However, the problem with Kodiyak has more to do with the user interface than with the lack of high-level data types. Maps are sufficient for implementing the MDOAG but they are cumbersome to use. They are generally "loaded" in one location of the parse tree and applied in another location some "distance" away. Consequently, even though a map may be completely loaded at the base on the far right side of the parse tree and only be applied at the base on the far left side of the parse tree, a large number of attributes and semantic functions must be used to pass the map from the right side of the tree to the left. This requires a great deal of repetitive code for a simple application. A better user interface would be one which:

1. Presents the BNF to the user graphically in the form of a tree.
2. Allows the user to traverse the tree using a mouse. Since most trees will not fit on the screen, it should bring up subtrees as the user traverses the tree.
3. Allows the user to "click" on a node in tree opening a window for attribute and semantic function specification.
4. Provides the user with a list of current attributes and semantic functions which could be selected by mouse and then applied at any desired location in the tree by pointing and clicking. This would serve as a rapid copying mechanism.

Such an interface should greatly reduce the time required to code a translation and provide a more intuitive view of the translation.

D. FURTHER WORK REQUIRED

A great deal of work remains to done to complete the MDOAG for the complete Spec language. This prototype provides something concrete to look at, tear apart, improve, and rebuild, if desired. Guidelines for extending the system have been provided in Chapter V.

An alternative methodology not mentioned in Chapter V but which may have merit is to translate Spec modules into concrete interfaces with assertion annotations in an annotation language like "ANNA" [Ref. 17]. The automatically generated module interfaces are completed by the module implementor. Then the Ada test facilities already provided by those languages may be used to test the implementation.

APPENDIX A

MODULE DRIVER AND OUTPUT ANALYZER GENERATOR CODE

```
! version stamp $Header: check.k,v 1.36 90/06/16 17:33:13 depasqua Locked $

! In the grammar, comments go from a "!" to the end of the line.
! Terminal symbols are entirely upper case or enclosed in single quotes (').
! Nonterminal symbols are entirely lower case.
! Lexical character classes start with a captial letter and are enclosed in {}.
! In a regular expression, x+ means one or more x's.
! In a regular expression, x* means zero or more x's.
! In a regular expression, [xyz] means x or y or z.
! In a regular expression, [^xyz] means any character except x or y or z.
! In a regular expression, [a-z] means any character between a and z.
! In a regular expression, . means any character except newline.

! definitions of lexical classes

%defineDigit            :[0-9]
%defineInt              :{Digit}+
%defineLower            :[a-z]
%defineUpper            :[A-Z]
%defineLetter           :({Lower}|{Upper})
%defineAlpha            :({Lower}|{Digit}|"_")
%defineBlank            :[ \t\n]
%define Quote          :["]
%define Backslash      :"\\"
%defineChar             :([^"\\]|{Backslash}{Quote}|{Backslash}{Backslash})
%define Op1            :("&"|"|"|"~"|"=>"|"<=>")
%define Op2            :("<"|">"|"="|"<="|">=")
%define Op3            :("~="|"~<"|"~>"|"~<="|"~>="|"=="|"~==")
%define Op4            :("+"|"-"|"*"|"/"|{Backslash}|MOD|"^")
%define Op5            :(U|IN|".."|"|"|"."|"[")
%define Op             :({Op1}|{Op2}|{Op3}|{Op4}|{Op5})

! definitions of white space and comments

                        :{Blank}+
                        :"--".*"\n"

! definitions of compound symbols and keywords

AND                     :"&"
OR                      :"|"
NOT                     :"~"
```

```
IMPLIES                 :"=>"
IFF                     :"<=>"

LE                      :"<="
GE                      :">="
NE                      :"~="
NLT                     :"~<"
NGT                     :"~>"
NLE                     :"~<="
NGE                     :"~>="
EQV                     :"=="
NEQV                    :"~=="

RANGE                   :".."
APPEND                  :"||"
MOD                     :{Backslash}|MOD
EXP                     :"^"

BIND                    :"      :      :"
ARROW                   :"->"

IF                      :IF
THEN                    :THEN
ELSE                    :ELSE
IN                      :IN
U                       :U
SUCH                    :SUCH{Blank}*THAT
ELSE_IF                 :ELSE{Blank}*IF

AS                      :AS
CHOOSE                  :CHOOSE
CONCEPT                 :CONCEPT
DEFAULT                 :DEFAULT
DEFINITION              :DEFINITION
DELAY                   :DELAY
DO                      :DO
END                     :END
EXCEPTION               :EXCEPTION
EXPORT                  :EXPORT
FI                      :FI
FOREACH                 :FOREACH
FROM                    :FROM
FUNCTION                :FUNCTION
GENERATE                :GENERATE
HIDE                    :HIDE
IMPORT                  :IMPORT
INHERIT                 :INHERIT
INITIALLY               :INITIALLY
INSTANCE                :INSTANCE
INVARIANT               :INVARIANT
MACHINE                 :MACHINE
```

```
MESSAGE                 :MESSAGE
MODEL                   :MODEL
OD                      :OD
OF                      :OF
OTHERWISE               :OTHERWISE
PERIOD                  :PERIOD
PRAGMA                  :PRAGMA
RENAME                  :RENAME
REPLY                   :REPLY
SEND                    :SEND
STATE                   :STATE
TEMPORAL                :TEMPORAL
TIME                    :TIME
TO                      :TO
TRANSACTION             :TRANSACTION
TRANSITION              :TRANSITION
TYPE                    :TYPE
VALUE                   :VALUE
VIRTUAL                 :VIRTUAL
WHEN                    :WHEN
WHERE                   :WHERE

QUANTIFIER              :{Upper}{Upper}+
NAME                    :(({Letter}{Alpha}*)|({Quote}{Op}{Quote}))
INTEGER_LITERAL         :{Int}
REAL_LITERAL            :{Int}"."{Int}
CHAR_LITERAL            :"'"."'"
STRING_LITERAL          :{Quote}{Char}*{Quote}

! operator precedences, %left means 2+3+4 is (2+3)+4.

%left                   ';', IF, DO, EXCEPTION, NAME, SEMI;
%left                   ',', COMMA;
%left                   SUCH;
%left                   '@';
%left                   IFF;
%left                   IMPLIES;
%left                   OR;
%left                   AND;
%left                   NOT;
%left                   '<', '>', '=', LE, GE, NE, NLT, NGT, NLE, NGE, EQV, NEQV;
%left                   IN, RANGE;
%left                   U, APPEND;
%left                   '+', '-', PLUS, MINUS;
%left                   '*', '/', MUL, DIV, MOD;
%left                   UMINUS;
%left                   EXP;
%left                   '$', '[', '(', '{', '.', DOT, WHERE;
%left                   STAR;

%%
```

```
!attribute declarations
start{
        main:string;
        main_pkg:string;
        check_pkg:string;
        driver:string;
        report:string;
        clist:string;
        q_iterator_macros:string;
        cond_type_pkg:string;
};
spec{
        debug:string;
        main:string;
        main_pkg:string;
        check_pkg:string;
        driver:string;
        report:string;
        clist:string;
        driver_gen_macro:string;
        q_iterator_macros:string;
        cond_type_pkg:string;
};
module{
        debug:string;
        main:string;
        main_pkg:string;
        check_pkg:string;
        driver:string;
        report:string;
        clist:string;
        driver_gen_macro:string;
        q_iterator_macros:string;
        cond_type_pkg:string;
};
function{
        debug:string;
        ada_interface_type:string;
        call:string;
        call_specs:string;
        call_actuals:string;
        actual_parms:string;

        fm_call_specs:string;
        r_call_specs:string;
        r_call_actuals:string;
        fm_call_actuals:string;
        r_parm_count:int;
        update_count:int;
        init_statements:string;
```

```
        main:string;
        main_pkg:string;
        check_pkg:string;
        driver:string;
        report:string;
        cond_type_pkg:string;
        driver_gen_macro:string;
        q_iterator_macros:string;

        instantiations:string;
        g_formal_part:string;
        g_parm_decls:string;
        g_obj_decls:string;
        g_actual_parms:string;
        clist:string;
        driver_basic_decl:string;
        mpkg_gets:string;
        q_with_clauses:string;
        condition_types:string;

};
module_header{
        g_parm_decls:string;
        function_designator:string;
        g_actual_parms:string;
        mpkg_gets:string;

};
pragmas{
        update:string->string;
        remove:string->string;
        update_count:int;
        init_statements:string;
};
formal_message{
        fm_call_specs:string;
        update:string->string;
        fm_call_actuals:string;

        fm_parm_specs:string;
        RE_fm_actual_parms:string;
        r_type_mark:string;
        gen_loop_vars:string;
        rpkg_puts:string;
        clist:string;
 };
 actual_message{
        rpkg_puts:string;
        r_actual_parms:string;
        remove:string->string;
        r_call_specs:string;
```

```
        r_call_actuals:string;
        r_parm_count:int;

        r_parm_spec:string;
        r_parm:string;
        RE_r_actual_parm:string;
        exception_trans:string;
        err_msg_when:string;
        RE_actual_parms:string;
        r_type_mark:string;
        exception_when_clauses:string;
        reply_exceptions:string;
};
where{
        where_expr_list_trans:string;
        c_seq_stmts:string;
        cr_parm:string;
        r_parm:string;
        err_msg_when:string;
        RE_actual_parms:string;
        quantifier_functions:string;
        q_iterator_macros:string;
        q_with_clauses:string;
};

optional_exception{
        flag:string;
};

messages{
        r_actual_parms:string;
        fm_call_specs:string;
        r_call_specs:string;
        r_call_actuals:string;
        fm_call_actuals:string;
        r_parm_count:int;
        update_count:int;
        init_statements:string;

        parm_specs:string;
        resp_trans:string;
        RE_actual_parms:string;
        fm_actual_parms:string;
        fm_parm_specs:string;
        g_parm_decls:string;
        g_actual_parms:string;
        r_type_mark:string;
        gen_loop_vars:string;
        exception_when_clauses:string;
        rpkg_puts:string;
        clist:string;
```

```
        mpkg_gets:string;
        quantifier_functions:string;
        q_with_clauses:string;
        q_iterator_macros:string;
        reply_exceptions:string;
};
message{
        r_actual_parms:string;
        fm_call_specs:string;
        r_call_specs:string;
        r_call_actuals:string;
        fm_call_actuals:string;
        r_parm_count:int;
        update_count:int;
        init_statements:string;

        parm_specs:string;
        resp_trans:string;
        RE_actual_parms:string;
        fm_actual_parms:string;
        fm_parm_specs:string;
        r_type_mark:string;
        gen_loop_vars:string;
        exception_when_clauses:string;
        rpkg_puts:string;
        clist:string;
        quantifier_functions:string;
        q_with_clauses:string;
        reply_exceptions:string;
        q_iterator_macros:string;
};
response{
        rpkg_puts:string;
        r_actual_parms:string;
        remove:string->string;
        r_call_specs:string;
        r_call_actuals:string;
        r_parm_count:int;

        r_parm_spec:string;
        resp_trans:string;
        RE_r_actual_parm:string;
        RE_actual_parms:string;
        r_type_mark:string;
        exception_when_clauses:string;
        quantifier_functions:string;
        q_with_clauses:string;
        q_iterator_macros:string;
        reply_exceptions:string;
};
response_cases{
```

```
        exception_list:string->string;
        rpkg_puts:string;
        r_actual_parms:string;
        remove:string->string;
        r_call_specs:string;
        r_call_actuals:string;
        r_parm_count:int;

        r_parm_spec:string;
        resp_cases_trans:string;
        class:string;
        RE_r_actual_parm:string;
        RE_actual_parms:string;
        r_type_mark:string;
        exception_when_clauses:string;
        quantifier_functions:string;
        q_with_clauses:string;
        q_iterator_macros:string;
        reply_exceptions:string;
};
response_set{
        rpkg_puts:string;
        r_actual_parms:string;
        remove:string->string;
        r_call_specs:string;
        r_call_actuals:string;
        r_parm_count:int;

        r_parm_spec:string;
        resp_set_trans:string;
        err_msg_when:string;
        RE_r_actual_parm:string;
        RE_actual_parms:string;
        r_type_mark:string;
        exception_when_clauses:string;
        parent_production:string;
        quantifier_functions:string;
        q_with_clauses:string;
        q_iterator_macros:string;
        reply_exceptions:string;
};
reply{
        rpkg_puts:string;
        r_actual_parms:string;
        remove:string->string;
        r_call_specs:string;
        r_call_actuals:string;
        r_parm_count:int;

        r_parm_spec:string;
        resp_set_trans:string;
```

```
        err_msg_when:string;
        RE_r_actual_parm:string;
        RE_actual_parms:string;
        r_type_mark:string;
        exception_when_clauses:string;
        quantifier_functions:string;
        q_with_clauses:string;
        q_iterator_macros:string;
        reply_exceptions:string;
};
concepts{
        c_subprog_specs:string;
        c_subprog_bodies:string;
        g_parm_decls:string;
        g_actual_parms:string;
        mpkg_gets:string;
        quantifier_functions:string;
        q_with_clauses:string;
        q_iterator_macros:string;
};
concept{
        c_subprog_spec:string;
        c_subprog_body:string;
        quantifier_functions:string;
        q_with_clauses:string;
        q_iterator_macros:string;



};
formal_name{
        g_parm_decls:string;
        c_designator:string;
        RE_fm_actual_parms:string;
        function_designator:string;
        g_actual_parms:string;
        mpkg_gets:string;
};
formal_arguments{
        r_actual_parms:string;
        fm_call_specs:string;
        update:string->string;
        remove:string->string;
        r_call_specs:string;
        r_call_actuals:string;
        fm_call_actuals:string;
        r_parm_count:int;

        fm_parm_specs:string;
        r_parm_spec:string;
        r_parm:string;
```

```
        RE_fm_actual_parms:string;
        RE_r_actual_parm:string;
        r_type_mark:string;
        gen_loop_vars:string;
        rpkg_puts:string;
        clist:string;

};
formals{
        r_actual_parms:string;
        fm_call_specs:string;
        update:string->string;
        remove:string->string;
        r_call_specs:string;
        r_call_actuals:string;
        fm_call_actuals:string;
        r_parm_count:int;

        g_parm_decls:string;
        r_parm_spec:string;
        q_parm_specs:string;
        fm_parm_specs:string;
        c_parm_specs:string;
        cr_type_mark:string;
        c_decl_part:string;
        cr_parm:string;
        r_parm:string;
        RE_fm_actual_parms:string;
        RE_r_actual_parm:string;
        r_type_mark:string;
        g_actual_parms:string;
        gen_loop_vars:string;
        rpkg_puts:string;
        clist:string;
        mpkg_gets:string;
        such_quantifier_trans:string;
        text:string;
};
        field_list{
        r_actual_parms:string;
        fm_call_specs:string;
        update:string->string;
        remove:string->string;
        r_call_specs:string;
        r_call_actuals:string;
        fm_call_actuals:string;
        r_parm_count:int;

        g_parm_decls:string;
        r_parm_spec:string;
        fm_parm_specs:string;
```

```
        q_parm_specs:string;
        c_parm_specs:string;
        cr_type_mark:string;
        c_parm_decls:string;
        cr_parm:string;
        r_parm:string;
        RE_fm_actual_parms:string;
        RE_r_actual_parm:string;
        r_type_mark:string;
        g_actual_parms:string;
        gen_loop_vars:string;
        rpkg_puts:string;
        clist:string;
        mpkg_gets:string;
        text:string;
};
type_binding{
        r_actual_parms:string;
        fm_call_specs:string;
        update:string->string;
        remove:string->string;
        r_call_specs:string;
        r_call_actuals:string;
        fm_call_actuals:string;
        r_parm_count:int;

        g_parm_decl:string;
        r_parm_spec:string;
        fm_parm_spec:string;
        q_parm_specs:string;
        c_parm_spec:string;
        cr_type_mark:string;
        c_parm_decl:string;
        cr_parm:string;
        r_parm:string;
        RE_fm_actual_parm:string;
        RE_r_actual_parm:string;
        r_type_mark:string;
        g_actual_parms:string;
        gen_loop_vars:string;
        rpkg_puts:string;
        clist:string;
        mpkg_gets:string;
        text:string;
};
!type_mark required for forming gen_loop_vars in proper format
name_list{
        r_actual_parms:string;
        fm_call_specs:string;
        update:string->string;
        remove:string->string;
```

```
        r_call_specs:string;
        r_call_actuals:string;
        fm_call_actuals:string;
        r_parm_count:int;

        identifier_list:string;
        type_mark:string;
        gen_loop_vars:string;
        rpkg_puts:string;
        mpkg_gets:string;
        text:string;
};
restriction{
        such_quantifier_trans:string;
        text:string;
};
optional_actual_name{
        text:string;
};
actual_name{
        type_mark:string;
        identifier:string;
        text:string;


};
actuals{
        update:string->string;
        init_statements:string;
        remove_parm:string;

        actual_parms:string;
        r_parm:string;
        text:string;
};
arg{
        actual_parm:string;
        r_parm:string;
        text:string;



};
expression_list{
        when_expr_list_trans:string;
        where_expr_list_trans:string;
        c_where_expr_list_trans:string;
        cr_parm:string;
        r_parm:string;
        in_err_msg_when:string;
        out_err_msg_when:string;
```

```
        text:string;
        RE_actual_parms:string;
        quantifier_functions:string;
        q_with_clauses:string;
        q_iterator_macros:string;
};
! expr_trans for when and general? case
! where_expr_trans for where expression transformation
expression{
        type_mark:string;
        expr_trans:string;
        cr_parm:string;
        r_parm:string;
        text:string;
        RE_actual_parms:string;
        clist:string;
        class:string;
        err_msg:string;
        conditional_trans:string;
        quantifier_functions:string;
        q_with_clauses:string;
        q_iterator_macros:string;
        is_leftmost:string;
};

middle_cases{
        err_msg:string;
        r_parm:string;
        cr_parm:string;
        RE_actual_parms:string;
        translation:string;
        quantifier_functions:string;
        q_with_clauses:string;
        q_iterator_macros:string;
};

literal{
        identifier:string;



};


AND{
        %text:string;
};
OR{
        %text:string;
};
NOT{
```

```
        %text:string;
};
IMPLIES{
        %text:string;
};
IFF{
        %text:string;
};

LE{
        %text:string;
};
GE{
        %text:string;
};
NE{
        %text:string;
};
NLT{
        %text:string;
};
NGT{
        %text:string;
};
NLE{
        %text:string;
};
NGE{
        %text:string;
};
EQV{
        %text:string;
};
NEQV{
        %text:string;
};
RANGE{
        %text:string;
};
APPEND{
        %text:string;
};
MOD{
        %text:string;
};
EXP{
        %text:string;
};

BIND{
        %text:string;
```

```
};
ARROW{
        %text:string;
};

IF{
        %text:string;
};
THEN{
        %text:string;
};
ELSE{
        %text:string;
};
IN{
        %text:string;
};
U{
        %text:string;
};
SUCH{
        %text:string;
};
ELSE_IF{
        %text:string;
};

AS{
        %text:string;
};
CHOOSE{
        %text:string;
};
CONCEPT{
        %text:string;
};
DEFINITION{
        %text:string;
};
DELAY{
        %text:string;
};
DO{
        %text:string;
};
END{
        %text:string;
};
EXCEPTION{
        %text:string;
};
```

```
EXPORT{
        %text:string;
};
FI{
        %text:string;
};
FOREACH{
        %text:string;
};
FROM{
        %text:string;
};
FUNCTION{
        %text:string;
};
GENERATE{
        %text:string;
};
HIDE{
        %text:string;
};
IMPORT{
        %text:string;
};
INHERIT{
        %text:string;
};
INITIALLY{
        %text:string;
};
INSTANCE{
        %text:string;
};
INVARIANT{
        %text:string;
};
MACHINE{
        %text:string;
};
MESSAGE{
        %text:string;
};
MODEL{
        %text:string;
};
OD{
        %text:string;
};
OF{
        %text:string;
};
```

```
OTHERWISE{
        %text:string;
};
PERIOD{
        %text:string;
};
RENAME{
        %text:string;
};
REPLY{
        %text:string;
};
SEND{
        %text:string;
};
STATE{
        %text:string;
};
TEMPORAL{
        %text:string;
};
TIME{
        %text:string;
};
TO{
        %text:string;
};
TRANSACTION{
        %text:string;
};
TRANSITION{
        %text:string;
};
TYPE{
        %text:string;
};
VALUE{
        %text:string;
};
VIRTUAL{
        %text:string;
};
WHEN{
        %text:string;
};
WHERE{
        %text:string;
        quantifier_functions:string;
        q_with_clauses:string;
        q_iterator_macros:string;
};
```

```
QUANTIFIER{
        %text:string;
        %line:int;
        all_checking_trans:string;
        some_checking_trans:string;
};
NAME{
        %text:string;
};
INTEGER_LITERAL{
        %text:string;
};
REAL_LITERAL{
        %text:string;
};
CHAR_LITERAL{
        %text:string;
};
STRING_LITERAL{
        %text:string;
};

%%
! productions of the grammar
!start
start
        : spec
   {
!disabled           %outfile("debug",spec.debug);
                    %outfile("main.a",spec.main);
                    %outfile("main_pkg.a",spec.main_pkg);
                    %outfile("check_pkg.m4",spec.check_pkg);
                    %outfile("driver.m4",spec.driver);
                    %outfile("report.a",spec.report);
                    %outfile("input_generator.m4",spec.driver_gen_macro);
                    %outfile("iterators.m4",spec.q_iterator_macros);
                    %outfile("condition_type_pkg.a",spec.cond_type_pkg);
   }
;
!spec
spec
        : spec module
   {
                    spec[1].debug=module.debug;
                    spec[1].main=module.main;
                    spec[1].main_pkg=module.main_pkg;
                    spec[1].check_pkg=module.check_pkg;
                    spec[1].driver=module.driver;
                    spec[1].report=module.report;
                    spec[1].driver_gen_macro=module.driver_gen_macro;
```

```
                spec[1].q_iterator_macros=module.q_iterator_macros;
                spec[1].cond_type_pkg=module.cond_type_pkg;
    }
|
    {}
:
        ! A production with nothing after the "|" means the empty string
        ! is a legal replacement for the left hand side.
!module
module
        : definition
          {}
        | function
          {
                module.debug=function.debug;
                module.main=function.main;
                module.main_pkg=function.main_pkg;
                module.check_pkg=function.check_pkg;
                module.driver=function.driver;
                module.report=function.report;
                module.driver_gen_macro=function.driver_gen_macro;

                !add generator.m4 visibility the group of iterator shells
                module.q_iterator_macros=
                    ["include(/n/suns2/work/student/depasqua/MACROS/",
                            "generator.m4)\n",
                    function.q_iterator_macros];

                module.cond_type_pkg=function.cond_type_pkg;
          }
        | type
          {}
        | machine
          {}
        | instance      ! of a generic module
          {}
        ;
!function
function
        : optionally_virtual FUNCTION module_header messages concepts
END
        {
!debug
function.debug=
    ["fm_call_specs\n\t\t",messages.fm_call_specs,"\n",
     "r_call_specs\n\t\t",messages.r_call_specs,"\n",
     "r_call_actuals\n\t\t",messages.r_call_actuals,"\n",
     "fm_call_actuals\n\t\t",messages.fm_call_actuals,"\n",
     "r_parm_count\n\t\t",i2s(messages.r_parm_count),"\n",
     "update_count\n\t\t",i2s(messages.update_count),"\n",
     "init_statements\n\t\t",messages.init_statements,"\n",
```

```
    "interface type is ",function.ada_interface_type,"\n",
    "function.g_formal_part is ",function.g_formal_part,"\n"];

!main
!procedure main (fixed) included so that all files
!necessary for ADA compilation would be generated
function.main=[
   "with REPORT;\n",
        "with MAIN_PKG;\n",
   "procedure MAIN is\n",
   "begin\n",
      "\tREPORT.OPEN;\n",
      "\twhile not (MAIN_PKG.TESTS_COMPLETE) loop\n",
         "\t\tMAIN_PKG.GET_TEST_PARAMETERS;\n",
         "\t\tMAIN_PKG.EXECUTE_TEST;\n",
      "\tend loop;\n",
      "\tREPORT.CLOSE;\n",
   "end MAIN;"];
!main_pkg
!main_pkg template
function.main_pkg=[
"package MAIN_PKG is\n",
   "\tfunction TESTS_COMPLETE return boolean;\n",
   "\tprocedure GET_TEST_PARAMETERS;\n",
   "\tprocedure EXECUTE_TEST;\n",
"end MAIN_PKG;\n\n",

"with FLT_IO;\n",
"with DRIVER;\n",
"with IMPLEMENTATION;\n",
"with TEXT_IO; use TEXT_IO;\n",
"package body MAIN_PKG is\n",
"\tINFILE:  FILE_TYPE;\n",
"\tASSURANCE:    FLOAT range 0.0..1.0;\n",
function.g_obj_decls,"\n\n",

"\tfunction TESTS_COMPLETE return boolean is\n",
"\tbegin\n",
   "\t\tif IS_OPEN(INFILE) and then END_OF_FILE(INFILE) then\n",
      "\t\t\tCLOSE(INFILE);\n",
      "\t\t\treturn TRUE;\n",
   "\t\telsif IS_OPEN(INFILE) then\n",
      "\t\t\treturn FALSE;\n",
   "\t\telse  OPEN(INFILE,IN_FILE,\"test_parameters\");\n",
        "\t\t\treturn END_OF_FILE(INFILE);\n",
   "\t\tend if;\n",
"\tend TESTS_COMPLETE;\n\n",

"\tprocedure GET_TEST_PARAMETERS is\n",
"\tbegin\n",
   "\t\tFLT_IO.GET(INFILE,ASSURANCE);\n",
```

```
    function.mpkg_gets,
"\tend GET_TEST_PARAMETERS;\n\n",

"\tprocedure EXECUTE_TEST is\n",
    function.driver_basic_decl,
"\tbegin\n",
    "\t\tNEW_DRIVER(ASSURANCE);\n",
"\tend EXECUTE_TEST;\n",
"end MAIN_PKG;\n"];

function.g_obj_decls=
function.g_parm_decls==""
> ""
#    [function.g_parm_decls,";\n"];

function.mpkg_gets=
[module_header.mpkg_gets,
  messages.mpkg_gets,
  concepts.mpkg_gets];

function.driver_basic_decl=
function.g_parm_decls==""
> ["\t\tprocedure NEW_DRIVER(assurance: float) ",
        "renames DRIVER;\n"]
#   ["\t\tprocedure NEW_DRIVER is new DRIVER(",
          function.g_actual_parms,");\n"];
  !check_pkg
!check package template output as check_pkg.a
function.check_pkg=[
    "include(/n/suns2/work/student/depasqua/MACROS/",
      "generator.m4)\n",
    "with REPORT;  use REPORT;\n",
    "with IMPLEMENTATION; use IMPLEMENTATION;\n",
    "with MDOAG_LIB; use MDOAG_LIB;\n",
    "with CONDITION_TYPE_PKG;  use CONDITION_TYPE_PKG;\n",
    "generic\n",
       "\tassurance:  float;\n",
       function.g_obj_decls,
    "package CHECK_PKG is\n",
       "\tprocedure CHECK(condition:  condition_type;\n",
            messages.parm_specs,");\n",
    "end CHECK_PKG;\n\n",
    function.q_with_clauses,
    "package body CHECK_PKG is\n\n",
       concepts.c_subprog_specs,"\n\n",
       "\tprocedure CHECK(condition:  condition_type;\n",
            messages.parm_specs,") is\n",
          "\t\tpreconditions_satisfied: boolean := false;\n",
          messages.quantifier_functions,
       "\tbegin\n",
          messages.resp_trans,
```

```
      "\tend CHECK;\n\n",
      concepts.c_subprog_bodies,
   "end CHECK_PKG;\n"];

function.g_formal_part=
function.g_parm_decls==""
> ""
#   ["generic\n",
function.g_parm_decls,";\n"];

!qunatifiers limited to only message responses
!and concepts
function.q_with_clauses=
   [messages.q_with_clauses,concepts.q_with_clauses];

  !logic required for proper delimeter (;) placement
!based on the existence or non-existence of generic
!parameter contributions of non-terminal productions
function.g_parm_decls=
   ~(module_header.g_parm_decls=="")
   >  ~(messages.g_parm_decls=="")
         >  ~(concepts.g_parm_decls=="")
               >  [module_header.g_parm_decls,";",
           messages.g_parm_decls,";",
           concepts.g_parm_decls]
   #    [module_header.g_parm_decls,";",
           messages.g_parm_decls]
         #    ~(concepts.g_parm_decls=="")
   >  [module_header.g_parm_decls,";",
           concepts.g_parm_decls]
   #    [module_header.g_parm_decls]
                        #    ~(messages.g_parm_decls=="")
   >  ~(concepts.g_parm_decls=="")
         >    [messages.g_parm_decls,";",
                  concepts.g_parm_decls]
         #  messages.g_parm_decls
   #    concepts.g_parm_decls;
!driver
!driver template output as driver.a
function.driver=[
   "include(/n/suns2/work/student/depasqua/MACROS",
     "/generator.m4)\n",
   function.g_formal_part,
   "procedure DRIVER(assurance:  in float);\n",
   "\n",
   "with GENERATOR;\n",
   "with CHECK_PKG;\n",
   "with REPORT;  use REPORT;\n",
   "with IMPLEMENTATION;  use IMPLEMENTATION;\n",
   "with CONDITION_TYPE_PKG; use CONDITION_TYPE_PKG;\n",
   "\n",
```

```
   "procedure DRIVER(assurance: in float) is\n",
      "\t","condition:  condition_type := normal;\n",
      messages.parm_specs,";\n",
      function.instantiations,
   "begin\n",
      "\t","REPORT.WRITE_INSTANCE_HEADER;","\n",
      "\t","foreach([",messages.gen_lcop_vars,"], GENERATOR,",
"[assurance], [","\n",
"\t\t","begin","\n",
      function.call,
   "\t\t\t","condition := normal;","\n",
   "\t\t","exception","\n",
   messages.exception_when_clauses,"\n",
   "\t\t\t","when others =>","\n",
      "\t\t\t\t","condition := ",
"unspecified_exception;","\n",
"\t\t","end;","\n",
"\t\t","BLACK_BOX.CHECK(condition,","\n",
   function.actual_parms,");\n",
"\t\t","INCREMENT_SAMPLES_TESTED;])","\n",
      "\t","REPORT.WRITE_INSTANCE_STATS;","\n",
   "end DRIVER;"];

!instantiations
function.instantiations=
function.g_formal_part==""
> (function.ada_interface_type=="function"
    > ["\tfunction IMPLEMENT(",messages.fm_parm_specs,")",
   " return ", messages.r_type_mark," renames ",
   module_header.function_designator,";","\n",
         "\tpackage BLACK_BOX is new CHECK_PKG(assurance);","\n"]
    # ["\tprocedure IMPLEMENT(",function.call_specs,")",
   " renames IMPLEMENTATION.",
   module_header.function_designator,";\n",
        "\tpackage BLACK_BOX is new CHECK_PKG(assurance);\n"])
#  (function.ada_interface_type == "function"
    > ["\tfunction IMPLEMENT is new ",
   module_header.function_designator,"(",
   module_header.g_actual_parms,");","\n",
         "\tpackage BLACK_BOX is new CHECK_PKG (assurance, ",
   function.g_actual_parms,");","\n"]
    # ["\tprocedure IMPLEMENT is new ",
   module_header.function_designator,"(",
      module_header.g_actual_parms,");\n",
         "\tpackage BLACK_BOX is new CHECK_PKG (assurance, ",
   function.g_actual_parms,");\n"]);

!ada_interface_type
!missing the case when there are 0 return parms
function.ada_interface_type=
   ((messages.r_parm_count == 1) &&
```

```
     (messages.update_count == 0))
>  "function"
#    "procedure";

!call_specs
! two cases for complete format coverage:
! case 1:  "" || "" -> "", "" || x -> x, x || "" -> x
! case 2:  x && y -> x ; y
function.call_specs=
    ((messages.fm_call_specs == "") ||
      (messages.r_call_specs == ""))
> [messages.fm_call_specs,messages.r_call_specs]
#   [messages.fm_call_specs,"; ",messages.r_call_specs];

!call
!function.call: calls the module being tested.
function.call=
    (function.ada_interface_type=="function")
    > [messages.r_call_actuals," := IMPLEMENT(",
          messages.fm_call_actuals,");\n"]
    #   [messages.init_statements,"\t\t\t",
          "IMPLEMENT(",function.call_actuals,");\n"];

!call_actuals
! see function.call_specs comment.
function.call_actuals=
    ((messages.fm_call_actuals == "") ||
      (messages.r_call_actuals == ""))
> [messages.fm_call_actuals,messages.r_call_actuals]
#   [messages.fm_call_actuals,", ",messages.r_call_actuals];

!actual_parms - sent to check package to be checked.
! see function.call_specs comment.
function.actual_parms=
    ((messages.fm_actual_parms == "") ||
      (messages.r_actual_parms == ""))
> [messages.fm_actual_parms,messages.r_actual_parms]
#   [messages.fm_actual_parms,", ",messages.r_actual_parms];

!g_actual_parms
  !logic required for proper delimeter (,) placement
function.g_actual_parms=
    ~(module_header.g_actual_parms=="")
    >   ~(messages.g_actual_parms=="")
          >   ~(concepts.g_actual_parms=="")
                >   [module_header.g_actual_parms,",",
            messages.g_actual_parms,",",
            concepts.g_actual_parms]
    #     [module_header.g_actual_parms,",",
            messages.g_actual_parms]
          #     ~(concepts.g_actual_parms=="")
```

```
    > [module_header.g_actual_parms,",",
          concepts.g_actual_parms]
    #  [module_header.g_actual_parms]
                              #   ~(messages.g_actual_parms=="")
    > ~(concepts.g_actual_parms=="")
        >   [messages.g_actual_parms,",",
                concepts.g_actual_parms]
        #  messages.g_actual_parms
    #  concepts.g_actual_parms;

!report
function.report=[
"with TEXT_IO;  use TEXT_IO;\n",
"with IMPLEMENTATION;  use IMPLEMENTATION;\n",
"with CONDITION_TYPE_PKG;  use CONDITION_TYPE_PKG;\n",
"package REPORT is\n",
   "\tprocedure ERROR(condition:  condition_type;\n",
                              messages.parm_specs,";\n",
               "msg:  string);\n",
   "\tprocedure OPEN;\n",
   "\tprocedure WRITE_INSTANCE_HEADER;\n",
   "\tprocedure INCREMENT_SAMPLES_TESTED;\n",
   "\tprocedure WRITE_INSTANCE_STATS;\n",
   "procedure CLOSE;\n",
"end REPORT;\n\n",

"package body REPORT is\n",
   "\ttotal_instances:  integer := 0;\n",
   "\tinstance_samples_tested: integer := 0;\n",
   "\ttotal_samples_tested: integer := 0;\n",
   "\tinstance_errors: integer := 0;\n",
   "\ttotal_errors:  integer := 0;\n",
   "\toutfile:  FILE_TYPE;\n",
   "\tpackage INT_IO is new INTEGER_IO(integer);\n",
   "\tuse INT_IO;\n",
   "\tpackage CONDITION_IO is new ",
  "ENUMERATION_IO(CONDITION_TYPE);\n",
   "\tuse CONDITION_IO;\n\n",

   "\tprocedure OPEN is\n",
   "\tbegin\n",
      "\t\tCREATE(outfile, OUT_FILE, \"",
module_header.function_designator,".err\");\n",
      "\t\tfor i in 1..65 loop PUT(outfile,\"*\");",
         "end loop;\n",
      "\t\tNEW_LINE(outfile);\n",
      "\t\tPUT_LINE(outfile,\"",
         module_header.function_designator,
         " Test Results\");\n",
      "\t\tfor i in 1..65 loop ",
         "PUT(outfile,\"*\"); end loop;\n",
```

```
        "\t\tNEW_LINE(outfile);\n",
        "\t\tNEW_LINE(outfile);\n",
      "\tend OPEN;\n\n",

    "\tprocedure WRITE_INSTANCE_HEADER is\n",
      "\tbegin\n",
        "\t\ttotal_instances := total_instances + 1;\n",
        "\t\tinstance_errors := 0;\n",
        "\t\tinstance_samples_tested := 0;\n",
        "\t\tNEW_LINE(outfile);\n",
        "\t\tfor i in 1..65 loop ",
          "PUT(outfile,\"*\"); end loop;\n",
        "\t\tNEW_LINE(outfile);\n",
        "\t\tPUT(outfile,\"TEST NUMBER  \");\n",
        "\t\tPUT(outfile,total_instances);\n",
        "\t\tNEW_LINE(outfile);\n",
    "\tend WRITE_INSTANCE_HEADER;\n\n",

    "\tprocedure INCREMENT_SAMPLES_TESTED is\n",
    "\tbegin\n",
    "\tinstance_samples_tested := ",
      "instance_samples_tested + 1;\n",
    "\ttotal_samples_tested := ",
      "total_samples_tested + 1;\n",
    "\tend INCREMENT_SAMPLES_TESTED;\n\n",

    "\tprocedure ERROR(condition: CONDITION_TYPE;\n",
                                  "messages.parm_specs,";\n",
       "msg: string) is\n",
    "\tbegin\n",
      "\t\tinstance_errors := instance_errors + 1;\n",
      "\t\ttotal_errors := total_errors + 1;\n",
      "\t\tPUT(outfile,\"ERROR:  \");\n",
      "\t\tNEW_LINE(outfile);\n",
        "\t\tPUT(outfile,msg);\n",
      "\t\tNEW_LINE(outfile);\n",
      "messages.rpkg_puts,
      "\t\tPUT(outfile,\"Condition = \");\n",
      "\t\tPUT(outfile,condition);\n",
      "\t\tNEW_LINE(outfile);\n",
    "\tend ERROR;\n\n",

    "\tprocedure WRITE_INSTANCE_STATS is\n",
    "\tbegin\n",
      "\t\tNEW_LINE(outfile);\n",
      "\t\tfor i in 1..33 loop ",
        "PUT(outfile,\"* \"); end loop;\n",
      "\t\tNEW_LINE(outfile);\n",
      "\t\tPUT(outfile,\"INSTANCE SAMPLES TESTED:  \");\n",
      "\t\tPUT(outfile,instance_samples_tested);\n",
      "\t\tNEW_LINE(outfile);\n",
```

```
    "\t\tPUT(outfile,\"INSTANCE ERRORS FOUND:    \");\n",
    "\t\tPUT(outfile,instance_errors);\n",
    "\t\tNEW_LINE(outfile);\n",
    "\t\tfor i in 1..65 loop ",
       "PUT(outfile,\"*\"); end loop;\n",
    "\t\tNEW_LINE(outfile);\n",
"\tend WRITE_INSTANCE_STATS;\n\n",

"\tprocedure CLOSE is\n",
"\tbegin\n",
    "\t\tNEW_LINE(outfile);\n",
    "\t\tfor i in 1..65 loop ",
       "PUT(outfile,\"*\"); end loop;\n",
    "\t\tNEW_LINE(outfile);\n",
    "\t\tPUT(outfile,\"TOTAL TESTS CONDUCTED:  \");\n",
    "\t\tPUT(outfile,total_instances);\n",
    "\t\tNEW_LINE(outfile);\n",
    "\t\tPUT(outfile,\"TOTAL SAMPLES TESTED:    \");\n",
    "\t\tPUT(outfile,total_samples_tested);\n",
    "\t\tNEW_LINE(outfile);\n",
    "\t\tPUT(outfile,\"TOTAL ERRORS FOUND:      \");\n",
    "\t\tPUT(outfile,total_errors);\n",
    "\t\tNEW_LINE(outfile);\n",
    "\t\tfor i in 1..65 loop ",
       "PUT(outfile,\"*\"); end loop;\n",
    "\t\tCLOSE(outfile);\n",
"\tend CLOSE;\n",
"end REPORT;\n"];
!cond_type_pkg
function.cond_type_pkg=
   ["with TEXT_IO; use TEXT_IO;\n",
     "package CONDITION_TYPE_PKG is\n",
        "\ttype CONDITION_TYPE is",
           " (",function.condition_types,");\n",
        "\tpackage CONDITION_TYPE_IO is new ",
"ENUMERATION_IO(CONDITION_TYPE);\n",
      "end CONDITION_TYPE_PKG;"];

function.condition_types=
   (messages.reply_exceptions == "")
> "normal, unspecified_exception"
#  ["normal, unspecified_exception,\n",
       messages.reply_exceptions];
!driver gen macro
function.driver_gen_macro=
     ["include(/n/suns2/work/student/depasqua/MACROS",
        "/generator.m4)\n",
      "--**put with and use clauses here**--\n",
      "generator(GENERATOR, \[assurance:  float\], ",
      "\[",messages.gen_loop_vars,"\],\n",
     "\[is\n",
```

```
        "\t--**put required declarations here**--\n",
      "begin\n",
        "\t--**put statements to generate values here**--\n",
        "\tgenerate(--**put generated values here**-);\n",
        "\t--**put more statements here as required**--\n",
      "end GENERATOR;\])\n"];

function.q_iterator_macros=
    [messages.q_iterator_macros,concepts.q_iterator_macros];
    }
  ;
    ! Virtual modules are for inheritance only, never used directly.
!machine
machine
  : optionally_virtual MACHINE module_header state messages temporals transactions
  concepts END
    {}
  ;
!type
  type
  : optionally_virtual TYPE module_header model messages temporals transactions
  concepts END
    {}
  ;
!definition
  definition
  : DEFINITION module_header concepts END
    {}
  ;
!instance
  instance
  : INSTANCE module_header where foreach concepts END
    {}
  ;
    ! For making instances or partial instantiations of generic modules.
    ! The foreach clause allows defining sets of instances.
!module_header
  module_header
  : formal_name defaults inherits imports export pragmas
    {
  module_header.mpkg_gets=formal_name.mpkg_gets;
  module_header.g_parm_decls=formal_name.g_parm_decls;
  module_header.function_designator=
  formal_name.function_designator;
  module_header.g_actual_parms=
    formal_name.g_parm_decls=="";
  > ""
  #  formal_name.g_actual_parms;

    }
  ;
```

```
    ! This part describes the static aspects of a module's interface.
    ! The dynamic aspects of the interface are described in the messages.
    ! A module is generic iff it has parameters.
    ! The parameters can be constrained by a SUCH THAT clause.
    ! A module can inherit the behavior of other modules.
    ! A module can import concepts from other modules.
    ! A module can export concepts for use by other modules.
!pragmas
  pragmas
: pragmas PRAGMA actual_name '(' actuals ')'
   {
pragmas[1].update_count=
   (actual_name.text=="update")
   > pragmas[2].update_count + 1
   #  pragmas[2].update_count;

pragmas[1].init_statements=
   (actual_name.text=="update")
   > [pragmas[2].init_statements,actuals.init_statements]
   #  pragmas[2].init_statements;

pragmas[1].remove=
   (actual_name.text=="update")
> pragmas[2].remove +| {(actuals.remove_parm:"true")}
#  pragmas[2].remove;

pragmas[1].update=
   (actual_name.text=="update")
> pragmas[2].update +| actuals.update
#  pragmas[2].update;
   }
|
   {
pragmas.update_count=0;
pragmas.init_statements="";
pragmas.remove={(?:string:"false")};
pragmas.update={(?:string:"false")};
   }
;
!inherits
  inherits
  : inherits INHERIT actual_name hide renames
     {}
  |
     {}
  ;
     ! Ancestors are generalizations or simplified views of a module.
     ! A module inherits all of the behavior of its ancestors.
     ! Hiding a message or concept means it will not be inherited.
     ! Inherited components can be renamed to avoid naming conflicts.
!hide
```

```
  hide
  : HIDE name_list
      {}
  |
      {}
  ;
      ! Useful for providing limited views of an actor.
      ! Different user classes may see different views of a system.
      ! Messages and concepts can be hidden.
!renames
  renames
  : renames RENAME NAME AS NAME
      {}
  |
      {}
  ;
      ! Renaming is useful for preventing NAME conflicts when inheriting
      ! from multiple sources, and for adapting modules for new uses.
      ! The parameters, model and state components, messages, exceptions,
      ! and concepts of an actor can be renamed.
!imports
  imports
  : imports IMPORT name_list FROM actual_name
      {}
  |
      {}
  ;
!export
  export
  : EXPORT name_list
      {}
  |
      {}
  ;
  !messages.g_formal_part is not implemented yet.
  !messages.mpkg_gets not implemented yet.
  messages
  : messages message
      {
messages[1].r_actual_parms=message.r_actual_parms;
messages[1].fm_call_specs=message.fm_call_specs;
messages[1].init_statements=message.init_statements;
              messages[1].update_count=message.update_count;
messages[1].r_parm_count=message.r_parm_count;
messages[1].fm_call_actuals=message.fm_call_actuals;
  messages[1].r_call_actuals=message.r_call_actuals;
messages[1].r_call_specs=message.r_call_specs;

  messages[1].reply_exceptions=
    (messages[2].reply_exceptions == "")
  > message.reply_exceptions
```

```
#    [messages[2].reply_exceptions,".\n",
         message.reply_exceptions];

 messages[1].mpkg_gets="";
 messages[1].parm_specs=[message.parm_specs];
 messages[1].resp_trans=message.resp_trans;
 messages[1].RE_actual_parms=[message.RE_actual_parms];
 messages[1].fm_actual_parms=[message.fm_actual_parms];
 messages[1].g_parm_decls="";
 messages[1].g_actual_parms="";
 messages[1].fm_parm_specs=message.fm_parm_specs;
 messages[1].r_type_mark=message.r_type_mark;
 messages[1].gen_loop_vars=message.gen_loop_vars;
 messages[1].exception_when_clauses=
    message.exception_when_clauses;
 messages[1].rpkg_puts=message.rpkg_puts;
 messages[1].quantifier_functions=
    [messages[2].quantifier_functions,
      message.quantifier_functions];
 messages[1].q_iterator_macros=
    [messages[2].q_iterator_macros,
      message.q_iterator_macros];
messages[1].q_with_clauses=
    [messages[2].q_with_clauses,
      message.q_with_clauses];
      }
 |
    {
 messages[1].r_actual_parms="";
 messages[1].reply_exceptions="";
 messages[1].parm_specs="";
 messages[1].RE_actual_parms="";
 messages[1].fm_actual_parms="";
 messages[1].g_parm_decls="";
 messages[1].g_actual_parms="";
 messages[1].fm_parm_specs="";
 messages[1].r_type_mark="";
 messages[1].gen_loop_vars="";
 messages[1].exception_when_clauses="";
 messages[1].rpkg_puts="";
 messages[1].quantifier_functions="";
 messages[1].q_iterator_macros="";
 messages[1].q_with_clauses="";
    }
 ;
!message
 message
 : MESSAGE formal_message pragmas response
    {
 message.r_actual_parms=response.r_actual_parms;
 message.update_count=pragmas.update_count;
```

```
    message.fm_call_specs=formal_message.fm_call_specs;
    message.init_statements=pragmas.init_statements;
    response.remove=pragmas.remove;
    formal_message.update=pragmas.update;
    message.fm_call_actuals=formal_message.fm_call_actuals;
    message.r_call_actuals=response.r_call_actuals;
    message.r_call_specs=response.r_call_specs;
    message.r_parm_count=response.r_parm_count;

    message.reply_exceptions=
       response.reply_exceptions;
    message.q_with_clauses=response.q_with_clauses;
    message.parm_specs=
       [formal_message.fm_parm_specs,";",response.r_parm_spec];
    message.resp_trans=response.resp_trans;

    !logic for delimiter "," placement.  The actual parameters
    !are followed by the error message; hence, a trailing comma.
    message.RE_actual_parms=
     ((formal_message.RE_fm_actual_parms=="") &&
      (response.r_actual_parms==""))
     > ""
     #  ((formal_message.RE_fm_actual_parms=="") ||
          (response.r_actual_parms==""))
        > [formal_message.RE_fm_actual_parms,
             response.r_actual_parms,", "]
        # [formal_message.RE_fm_actual_parms,", ",
            response.r_actual_parms,", "];

    response.RE_actual_parms=message.RE_actual_parms;

    message.fm_actual_parms=formal_message.RE_fm_actual_parms;
    message.r_type_mark=response.r_type_mark;
    message.gen_loop_vars=formal_message.gen_loop_vars;
    message.exception_when_clauses=
       response.exception_when_clauses;
    message.fm_parm_specs=formal_message.fm_parm_specs;
    message.rpkg_puts=
       [formal_message.rpkg_puts,response.rpkg_puts];
    message.quantifier_functions=
       response.quantifier_functions;
    message.q_iterator_macros=
       response.q_iterator_macros;
       }
    ;
  !response
    response
    : response_set
       {
  response.rpkg_puts=response_set.rpkg_puts;
    response.r_actual_parms=response_set.r_actual_parms;
```

```
response_set.remove=response.remove;
response.r_call_specs=response_set.r_call_specs;
response.r_call_actuals=response_set.r_call_actuals;
response.r_parm_count=response_set.r_parm_count;

response.reply_exceptions=response_set.reply_exceptions;
response.q_with_clauses=response_set.q_with_clauses;
response.r_parm_spec=response_set.r_parm_spec;
response.resp_trans=response_set.resp_set_trans;
response.RE_r_actual_parm=response_set.RE_r_actual_parm;
response_set.RE_actual_parms=response.RE_actual_parms;
response.r_type_mark=response_set.r_type_mark;
response.exception_when_clauses=
   response_set.exception_when_clauses;
response_set.parent_production="response";
response.quantifier_functions=
   response_set.quantifier_functions;
response.q_iterator_macros=response_set.q_iterator_macros;

   }
| response_cases
   {
response.rpkg_puts=response_cases.rpkg_puts;
response.r_actual_parms=response_cases.r_actual_parms;
response_cases.remove=response.remove;
response.r_call_specs=response_cases.r_call_specs;
response.r_call_actuals=response_cases.r_call_actuals;
response.r_parm_count=response_cases.r_parm_count;

response.reply_exceptions=response_cases.reply_exceptions;
response.q_with_clauses=response_cases.q_with_clauses;
response.r_parm_spec=response_cases.r_parm_spec;
response.resp_trans=response_cases.resp_cases_trans;
response.RE_r_actual_parm=response_cases.RE_r_actual_parm;
response_cases.RE_actual_parms=response.RE_actual_parms;
response.r_type_mark=response_cases.r_type_mark;
response.exception_when_clauses=
   response_cases.exception_when_clauses;
response.quantifier_functions=
   response_cases.quantifier_functions;
response.q_iterator_macros=
   response_cases.q_iterator_macros;

   }
;
!response_cases
 response_cases
 : WHEN expression_list response_set pragmas response_cases
   {
 response_cases[1].rpkg_puts=
  (response_set.r_parm_count > 0)
```

```
  > response_set.rpkg_puts
  #  response_cases[2].rpkg_puts;

response_cases[1].r_actual_parms=
  (response_set.r_parm_count > 0)
  > response_set.r_actual_parms
  #  response_cases[2].r_actual_parms;
response_set.remove=response_cases[1].remove;
response_cases[2].remove=response_cases[1].remove;
response_cases[1].r_call_specs=
   (response_set.r_parm_count > 0)
              > response_set.r_call_specs
   #  response_cases[2].r_call_specs;
response_cases[1].r_call_actuals=
   (response_set.r_parm_count > 0)
              > response_set.r_call_actuals
   #  response_cases[2].r_call_actuals;
response_cases.r_parm_count=
   (response_set.r_parm_count > 0)
              > response_set.r_parm_count
   #  response_cases[2].r_parm_count;

!enumeration of condition types.
!if the exception from the response set is already in the
!list of exceptions then do not add it.  The empty string
!is "in" the list already, hence it is never added and
!does not impact on delimiter placement.  (See OTHERWISE
!response cases production where the map is initially
!loaded.  Graphically, view the process as starting at
!the OTHERWISE -> response_set, where the first exception
!(if it exists), the empty string (""), and "almost every-
!where false" are loaded to a map.  Then, additional
!exceptions are added from the WHEN -> SET CASES production
!where SET potentially has a new exception and CASES is
!is formed already (i.e., synthesized).
response_cases[1].reply_exceptions=
   (response_cases[2].exception_list(response_set.reply_exceptions) == "true")
   > response_cases[2].reply_exceptions
   #  [response_set.reply_exceptions,",\n",
         response_cases[2].reply_exceptions];
response_cases[1].exception_list=
   response_cases[2].exception_list +|
   {(response_set.reply_exceptions:"true")};

response_cases[1].q_with_clauses=
   [expression_list.q_with_clauses,
    response_set.q_with_clauses,
    response_cases[2].q_with_clauses];

response_cases[1].r_parm_spec=
   (response_set.r_parm_count > 0)
```

```
    > response_set.r_parm_spec
    #  response_cases[2].r_parm_spec;

response_cases[1].class="WHEN";

 response_cases[1].resp_cases_trans=
    ["\t\tif ",expression_list.when_expr_list_trans,
" then\n",
response_set.resp_set_trans,
"\t\t\tpreconditions_satisfied := true;\n",
      "\t\tend if;\n",
        response_cases[2].resp_cases_trans];

 expression_list.cr_parm="#";
 expression_list.r_parm="#";
 response_set.err_msg_when=expression_list.out_err_msg_when;

 response_cases[1].RE_r_actual_parm=
  [response_set.RE_r_actual_parm];

 response_cases[2].RE_actual_parms=
  [response_cases[1].RE_actual_parms];

 response_set.RE_actual_parms=
  response_cases[1].RE_actual_parms;

 !see comment above concerning one return parm.
 response_cases.r_type_mark=response_set.r_type_mark;

 response_cases[1].exception_when_clauses=
    [response_set.exception_when_clauses,
      response_cases[2].exception_when_clauses];

 response_set.parent_production="response_cases";

 response_cases[1].quantifier_functions=
    [expression_list.quantifier_functions,
      response_cases[2].quantifier_functions,
      response_set.quantifier_functions];

 response_cases[1].q_iterator_macros=
    [expression_list.q_iterator_macros,
      response_cases[2].q_iterator_macros,
      response_set.q_iterator_macros];
    }
 | OTHERWISE response_set pragmas
    {
 response_cases.rpkg_puts=response_set.rpkg_puts;
 response_cases.r_actual_parms=response_set.r_actual_parms;
 response_set.remove=response_cases.remove;
 response_cases.r_parm_count=response_set.r_parm_count;
```

```
response_cases.r_call_specs=response_set.r_call_specs;
response_cases.r_call_actuals=response_set.r_call_actuals;

response_cases.reply_exceptions=
   response_set.reply_exceptions;

!the exception list is initialized here and is passed up.
!Initially, it is almost everywhere "false", contains the
!SET exception (if one exists, otherwise its ""), and
!contains the empty string ("") (See CASES -> SET CASES).
response_cases.exception_list=
   {(?:string:"false")} +|
   {(response_set.reply_exceptions:"true")} +|
   {("":"true")};
response_cases.q_with_clauses=response_set.q_with_clauses;
response_cases.r_parm_spec=response_set.r_parm_spec;

response_cases.class="OTHERWISE";

response_cases.resp_cases_trans=
   ["\t\tif not (preconditions_satisfied) then\n",
        response_set.resp_set_trans,
     "\t\tend if;\n"];
response_set.err_msg_when="OTHERWISE ";

response_cases.RE_r_actual_parm=
  response_set.RE_r_actual_parm;

response_set.RE_actual_parms=
  response_cases.RE_actual_parms;

response_cases.r_type_mark=response_set.r_type_mark;

response_cases.exception_when_clauses=
response_set.exception_when_clauses;

response_set.parent_production="response_cases";

response_cases.quantifier_functions=
   response_set.quantifier_functions;

response_cases.q_iterator_macros=
   response_set.q_iterator_macros;

   }
 ;
!response_set
 response_set
 : choose reply sends transition
   {
 response_set.rpkg_puts=reply.rpkg_puts;
```

```
  response_set.r_actual_parms=reply.r_actual_parms;
  reply.remove=response_set.remove;
  response_set.r_call_specs=reply.r_call_specs;
  response_set.r_call_actuals=reply.r_call_actuals;
  response_set.r_parm_count=reply.r_parm_count;


  response_set.reply_exceptions=
     reply.reply_exceptions;
  response_set.q_with_clauses=
     [reply.q_with_clauses];
  response_set.r_parm_spec=reply.r_parm_spec;
  response_set.resp_set_trans=reply.resp_set_trans;

  ! If "response" produces "response set",
  ! then no "when clause" exists.
  reply.err_msg_when=
     (response_set.parent_production == "response")
     > ""
     #  response_set.err_msg_when;

  response_set.RE_r_actual_parm=reply.RE_r_actual_parm;
  reply.RE_actual_parms=response_set.RE_actual_parms;
  response_set.r_type_mark=reply.r_type_mark;
  response_set.exception_when_clauses=
     reply.exception_when_clauses;

  response_set.quantifier_functions=
     reply.quantifier_functions;

  response_set.q_iterator_macros=
     reply.q_iterator_macros;
     }
  ;
!choose
  choose
  : CHOOSE '(' formals ')'
     {



     }
  |
     {


     }
  ;
!reply
reply
  : REPLY actual_message where
```

```
    {
reply.rpkg_puts=actual_message.rpkg_puts;
 reply.r_actual_parms=actual_message.r_actual_parms;
 actual_message.remove=reply.remove;
 reply.r_call_specs=actual_message.r_call_specs;
 reply.r_call_actuals=actual_message.r_call_actuals;
 reply.r_parm_count=actual_message.r_parm_count;

 reply.reply_exceptions=
    actual_message.reply_exceptions;
 reply.q_with_clauses=
    where.q_with_clauses;
 reply.r_parm_spec=actual_message.r_parm_spec;
 reply.resp_set_trans=
    [actual_message.exception_trans,
      where.where_expr_list_trans];
 where.r_parm=actual_message.r_parm;
 where.err_msg_when=reply.err_msg_when;

 reply.RE_r_actual_parm=actual_message.RE_r_actual_parm;
 where.RE_actual_parms=reply.RE_actual_parms;
 actual_message.RE_actual_parms=reply.RE_actual_parms;
 actual_message.err_msg_when=reply.err_msg_when;
 reply.r_type_mark=actual_message.r_type_mark;
 reply.exception_when_clauses=
    actual_message.exception_when_clauses;

 reply.quantifier_functions=
    where.quantifier_functions;

 reply.q_iterator_macros=
    where.q_iterator_macros;

 !flag indicating reply origin.
 where.cr_parm="#";
    }
 | GENERATE actual_message where       ! used in generators
    {



    }
 |
    {
 reply.rpkg_puts="";
 reply.r_actual_parms="";
 reply.r_call_specs="";
 reply.r_call_actuals="";
 reply.r_parm_count=0;

reply.q_with_clauses="";
```

```
reply.reply_exceptions="";

    }
  ;
!sends
  sends
  : sends send
    {


    }
  |
    {


    }
  ;
!send
  send
  : SEND actual_message TO actual_name where foreach
    {


    }
  ;
!transition
  transition
  : TRANSITION expression_list! for describing state changes
    {


    }
  |
    {


    }
  ;
!formal_message
  formal_message
  : optional_exception optional_formal_name formal_argument  defaults
    {
  formal_message.fm_call_specs=formal_arguments.fm_call_specs;
  formal_arguments.update=formal_message.update;
  formal_message.fm_call_actuals=
```

```
    formal_arguments.fm_call_actuals;

  formal_message.fm_parm_specs=formal_arguments.fm_parm_specs;

  formal_message.RE_fm_actual_parms=
    formal_arguments.RE_fm_actual_parms;

  formal_message.r_type_mark=formal_arguments.r_type_mark;
  formal_message.gen_loop_vars=formal_arguments.gen_loop_vars;
  formal_message.rpkg_puts=formal_arguments.rpkg_puts;

    }
  ;
!defaults
defaults
: DEFAULT expression_list
   {}
|%prec SEMI! must have a lower precedence than DEFAULT
   {}
;

!actual_message
  actual_message
  : optional_exception optional_actual_name formal_arguments
    {
actual_message.rpkg_puts=formal_arguments.rpkg_puts;
  actual_message.r_actual_parms=formal_arguments.r_actual_parms;
  formal_arguments.remove=actual_message.remove;
  actual_message.r_call_actuals=formal_arguments.r_call_actuals;
  actual_message.r_call_specs=formal_arguments.r_call_specs;
  actual_message.r_parm_count=formal_arguments.r_parm_count;

  actual_message.reply_exceptions=
     (optional_exception.flag == "true")
  > ["\t\t\' ,optional_actual_name.text,"_condition"]
  #  "";

  actual_message.r_parm_spec=formal_arguments r_parm_spec;
  actual_message.r_parm=formal_arguments.r_parm;

  actual_message.RE_r_actual_parm=
    formal_arguments.RE_r_actual_parm;

  actual_message.exception_trans=
     optional_exception.flag=="true"
  >["\t\t\t\tif not (condition = ",
       optional_actual_name.text,"_condition)  ",
       "then\n",
  "\t\t\t\t\tREPORT.ERROR(condition,\n",
  actual_message.RE_actual_parms,
  "\t\t\t\t\t\t\"",actual_message.err_msg_when,
```

```
  " NOT EXCEPTION ",
  optional_actual_name.text,"\");\n",
        "\t\t\t\tend if;\n"]
  # "";

  actual_message.exception_when_clauses=
     optional_exception.flag=="true"
  > ["\t\t\twhen ",optional_actual_name.text," =>","\n",
        "\t\t\t\tcondition := ",
          optional_actual_name.text,"_condition;","\n"]
  # "";
  actual_message.r_type_mark=formal_arguments.r_type_mark;
     }
  ;
!where
  where
  : WHERE expression_list
     {
  where.q_with_clauses=expression_list.q_with_clauses;
  where.where_expr_list_trans=
     expression_list.where_expr_list_trans;
  where.c_seq_stmts=expression_list.c_where_expr_list_trans;
  expression_list.cr_parm=where.cr_parm;
  expression_list.r_parm=where.r_parm;
  expression_list.in_err_msg_when=where.err_msg_when;

  expression_list.RE_actual_parms=where.RE_actual_parms;

  where.quantifier_functions=
     expression_list.quantifier_functions;

  where.q_iterator_macros=
     expression_list.q_iterator_macros;

     }
  |%prec SEMI! must have a lower precedence than WHERE
     {
  where.q_with_clauses="";
  where.where_expr_list_trans="";
  where.c_seq_stmts="";
  where.quantifier_functions="";
  where.q_iterator_macros="";
     }
  ;
!optionally_virtual
  optionally_virtual
  : VIRTUAL
     {}
  |
     {}
  ;
```

```
!optional_exception
  optional_exception
  : EXCEPTION
     {
  optional_exception.flag="true";


     }
  |%prec SEMI
     {

  optional_exception.flag="false";

     }
  ;
!foreach
  foreach
  : FOREACH '(' formals ')'
     {



     }
  |
     {



     }
  ;
     ! foreach is used to describe a set of messages or instances
!model
  model! data types have conceptual models for values
  : MODEL formal_arguments invariant pragmas
     {



     }
  |
     {



     }
  ;
!state
  state! machines have conceptual models for states
  : STATE formal_arguments invariant initially pragmas
     {}
  |
```

```
    {}
  ;
!invariant
  invariant! invariants are true for all states or instances
  : INVARIANT expression_list
    {}
  ;
!initially
  initially! initial conditions are true only at the beginning
  : INITIALLY expression_list
    {}
  ;
!temporals
  temporals
  : temporals temporal
    {



    }
  |
    {


    }
  ;
!temporal
  temporal
  : TEMPORAL formal_name defaults where response
    {



    }
  ;
    ! Temporal events are trigged at absolute times,
    ! in terms of the local clock of the actor.
    ! The "where" describes the triggering conditions
    ! in terms of TIME, PERIOD, and DELAY.
!transactions
  transactions
  : transactions transaction
    {



    }
  |
    {
```

```
        }
    ;
!transaction
    transaction
    : TRANSACTION formal_name '=' action_list where foreach
        {



        }
    ;
        ! Transactions are atomic.
        ! The where clause can specify timing constraints.
!action_list
    action_list
      action_list ';' action    %prec SEMI    ! sequence
        {



        }
    | action
        {



        }
    ;
!action
    action
    : action action    %prec STAR    ! unordered set of actions
      {



      }
| IF alternatives FI    ! choice
      {



      }
| DO alternatives OD    ! repeated choice
      {



      }
| actual_name    ! a normal message or subtransaction
      {
```

```
    }
| EXCEPTION actual_name     ! an exception message
    {



    }
;
!alternatives
alternatives
: alternatives OR guard action_list
    {



    }
| guard action_list
    {



    }
;
!guard
guard
: WHEN expression_list ARROW
    {



    }
|
    {



    }
;
!g_parm_decl not implemented yet.
!mpkg_gets unimplemented.
!concepts
concepts
: concepts concept
    {
concepts[1].q_with_clauses=
    [concepts[2].q_with_clauses,
      concept.q_with_clauses];
concepts[1].mpkg_gets="";
```

```
concepts[1].c_subprog_specs=
   [concepts[2].c_subprog_specs,concept.c_subprog_spec];
concepts[1].c_subprog_bodies=
   [concepts[2].c_subprog_bodies,concept.c_subprog_body];
concepts[1].g_parm_decls="";
concepts[1].g_actual_parms="";
concepts[1].q_iterator_macros=
   [concept.q_iterator_macros];


   }
|
   {
concepts[1].q_with_clauses="";
concepts[1].mpkg_gets="";
concepts[1].c_subprog_specs="";
concepts[1].c_subprog_bodies="";
   concepts[1].g_parm_decls="";
concepts[1].g_actual_parms="";
concepts[1].q_iterator_macros="";



   }
;
! r_parm="#" is a not from reply subtree flag.
!concept
concept
: CONCEPT formal_name ':' expression defaults where
   ! constants
   {
concept.q_with_clauses=where.q_with_clauses;
concept.c_subprog_spec="";
concept.c_subprog_body="";
where.r_parm="#";
concept.q_iterator_macros=where.q_iterator_macros;



   }
| CONCEPT formal_name '(' formals ')' defaults VALUE '(' formals ')' where
   ! functions, defined with preconditions and postconditions
   {
concept.q_with_clauses=
   where.q_with_clauses;
concept.c_subprog_spec=[
   "\tfunction ",formal_name.c_designator,
         "(",formals[1].c_parm_specs,") return ",
          formals[2].cr_type_mark,";\n"];
concept.c_subprog_body=[
   "\tfunction ",formal_name.c_designator,
```

```
"(",formals[1].c_parm_specs,") return ",
                formals[2].cr_type_mark," is\n",
            formals[2].c_decl_part,";",
    "\tbegin\n",
            where.c_seq_stmts,"\n",
    "\tend ",formal_name.c_designator,";\n\n"];
where.cr_parm=formals[2].cr_parm;
where.r_parm="#";

concept.q_iterator_macros=where.q_iterator_macros;



    }
;
!optional_formal_name
optional_formal_name
: formal_name
    {



    }
|
    {



    }
;
!formal_name
formal_name
: NAME '(' formals ')'
    {
formal_name.mpkg_gets=formals.mpkg_gets;
formal_name.g_parm_decls=formals.g_parm_decls;
    formal_name.function_designator=NAME.%text;
formal_name.g_actual_parms=formals.g_actual_parms;


    }
| NAME
    {
formal_name.mpkg_gets="";
formal_name.g_parm_decls="";
formal_name.c_designator=NAME.%text;
    formal_name.function_designator=NAME.%text;



    }
```

```
;
!formal_arguments
formal_arguments
: '(' formals ')'
    {
  formal_arguments.r_actual_parms=formals.r_actual_parms;
  formal_arguments.fm_call_specs=formals.fm_call_specs;
  formals.update=formal_arguments.update;
  formal_arguments.fm_call_actuals=
     formals.fm_call_actuals;

  formals.remove=formal_arguments.remove;
  formal_arguments.r_call_actuals=
     formals.r_call_actuals;
  formal_arguments.r_call_specs=
     formals.r_call_specs;
  formal_arguments.r_parm_count=formals.r_parm_count;

formal_arguments.fm_parm_specs=formals.fm_parm_specs;
formal_arguments.r_parm_spec=formals.r_parm_spec;
formal_arguments.r_parm=formals.r_parm;

formal_arguments.RE_fm_actual_parms=
  formals.RE_fm_actual_parms;

formal_arguments.RE_r_actual_parm=
  formals.RE_r_actual_parm;
formal_arguments.r_type_mark=formals.r_type_mark;
formal_arguments.gen_loop_vars=formals.gen_loop_vars;
formal_arguments.rpkg_puts=formals.rpkg_puts;
    }
|
    {
formal_arguments.r_actual_parms="";
formal_arguments.r_parm_count=0;
formal_arguments.r_call_specs="";
formal_arguments.r_call_actuals="";

formal_arguments.fm_parm_specs="";
formal_arguments.r_parm_spec="";
formal_arguments.r_parm="";
formal_arguments.RE_fm_actual_parms="";
formal_arguments.RE_r_actual_parm="";
formal_arguments.r_type_mark="";
formal_arguments.gen_loop_vars="";


    }
;
!formals
formals
```

```
: field_list restriction
    {
  formals.r_actual_parms=field_list.r_actual_parms;
  formals.fm_call_specs=field_list.fm_call_specs;
  field_list.update=formals.update;
  formals.fm_call_actuals=
     field_list.fm_call_actuals;

  field_list.remove=formals.remove;
  formals.r_call_actuals=
     field_list.r_call_actuals;
  formals.r_call_specs=
     field_list.r_call_specs;
  formals.r_parm_count=field_list.r_parm_count;

formals.mpkg_gets=field_list.mpkg_gets;
formals.g_parm_decls=field_list.g_parm_decls;
formals.fm_parm_specs=field_list.fm_parm_specs;
formals.r_parm_spec=field_list.r_parm_spec;
formals.c_parm_specs=field_list.c_parm_specs;
formals.q_parm_specs=field_list.q_parm_specs;
formals.cr_type_mark=field_list.cr_type_mark;
formals.c_decl_part=field_list.c_parm_decls;
formals.cr_parm=field_list.cr_parm;
formals.r_parm=field_list.r_parm;
    formals.text=[field_list.text," ",restriction.text];
formals.RE_fm_actual_parms=field_list.RE_fm_actual_parms;
formals.RE_r_actual_parm=field_list.RE_r_actual_parm;

formals.r_type_mark=field_list.r_type_mark;
formals.g_actual_parms=field_list.g_actual_parms;
formals.gen_loop_vars=field_list.gen_loop_vars;
formals.rpkg_puts=field_list.rpkg_puts;

formals.such_quantifier_trans=
    restriction.such_quantifier_trans;
    }
;
!Limitation:  Only 1 cr_type_mark allowed.
!field_list
field_list
: field_list ',' type_binding
    {
  field_list[1].r_actual_parms=
     [field_list[2].r_actual_parms,", ",
       type_binding.r_actual_parms];

                              !See comments at function.call_specs.
  field_list[1].fm_call_specs=
     ((field_list[2].fm_call_specs == "") ||
       (type_binding.fm_call_specs == ""))
```

```
> [field_list[2].fm_call_specs,type_binding.fm_call_specs]
#  [field_list[2].fm_call_specs,"; ",
       type_binding.fm_call_specs];
field_list[2].update=field_list[1].update;
type_binding.update=field_list[1].update;
field_list[1].fm_call_actuals=
   [field_list[2].fm_call_actuals,", ",
    type_binding.fm_call_actuals];

field_list[2].remove=field_list[1].remove;
type_binding.remove=field_list[1].remove;

!See call_specs comment at function.
field_list[1].r_call_actuals=
   (field_list[2].r_call_actuals == "" ||
    type_binding.r_call_actuals == "")
> [field_list[2].r_call_actuals,
       type_binding.r_call_actuals]
#  [field_list[2].r_call_actuals,", ",
       type_binding.r_call_actuals];

!S  all_specs comment at function.
fi   list[1].r_call_specs=
     eld_list[2].r_call_specs == "" ||
     type_binding.r_call_specs == "")
> [field_list[2].r_call_specs,
       type_binding.r_call_specs]
#  [field_list[2].r_call_specs,"; ",
       type_binding.r_call_specs];

field_list[1].r_parm_count=
   field_list[2].r_parm_count + type_binding.r_parm_count;

field_list[1].mpkg_gets=
   [field_list[2].mpkg_gets,type_binding.mpkg_gets];
field_list[1].g_parm_decls=
   [field_list[2].g_parm_decls,";",type_binding.g_parm_decl];
field_list[1].fm_parm_specs=
   [field_list[2].fm_parm_specs,";",type_binding.fm_parm_spec];

field_list[1].r_parm_spec=
   [field_list[2].r_parm_spec,";",type_binding.r_parm_spec];
field_list[1].q_parm_specs=
   [field_list[2].q_parm_specs,"; ",type_binding.q_parm_specs];
field_list[1].c_parm_specs=
   [field_list[2].c_parm_specs,";\n",type_binding.c_parm_spec];
field_list[1].cr_type_mark=type_binding.cr_type_mark;
field_list[1].c_parm_decls=
   [field_list[2].c_parm_decls,";",type_binding.c_parm_decl];
field_list.r_type_mark=type_binding.r_type_mark;
```

```
field_list[1].RE_fm_actual_parms=
  [field_list[2].RE_fm_actual_parms,",",
    type_binding.RE_fm_actual_parm];
field_list[1].RE_r_actual_parm=type_binding.RE_r_actual_parm;
field_list[1].g_actual_parms=
[field_list[2].g_actual_parms,",",
  type_binding.g_actual_parms];
field_list[1].gen_loop_vars=
[field_list[2].gen_loop_vars,";",
  type_binding.gen_loop_vars];
field_list[1].rpkg_puts=[field_list[2].rpkg_puts,
    type_binding.rpkg_puts];
field_list[1].text=
    [field_list[2].text," , ",type_binding.text];


    }
| type_binding
    {
field_list.r_actual_parms=type_binding.r_actual_parms;
type_binding.remove=field_list.remove;
type_binding.update=field_list.update;
field_list.r_call_actuals=type_binding.r_call_actuals;
field_list.r_call_specs=type_binding.r_call_specs;
field_list.r_parm_count=type_binding.r_parm_count;
field_list.fm_call_actuals=type_binding.fm_call_actuals;
field_list.fm_call_specs=type_binding.fm_call_specs;
field_list.mpkg_gets=type_binding.mpkg_gets;
field_list.g_parm_decls=[type_binding.g_parm_decl];
field_list.fm_parm_specs=[type_binding.fm_parm_spec];
field_list.r_parm_spec=[type_binding.r_parm_spec];
field_list.c_parm_specs=[type_binding.c_parm_spec];
field_list.q_parm_specs=[type_binding.q_parm_specs];
field_list.cr_type_mark=[type_binding.cr_type_mark];
field_list.c_parm_decls=[type_binding.c_parm_decl];
field_list.cr_parm=type_binding.cr_parm;
field_list.r_parm=type_binding.r_parm;

field_list.RE_fm_actual_parms=type_binding.RE_fm_actual_parm;
field_list.RE_r_actual_parm=type_binding.RE_r_actual_parm;
field_list.r_type_mark=type_binding.r_type_mark;
    field_list.g_actual_parms=type_binding.g_actual_parms;
field_list.gen_loop_vars=type_binding.gen_loop_vars;
field_list.rpkg_puts=type_binding.rpkg_puts;
field_list.text=type_binding.text;
    }
;
! identifier_list for c_parm_decl should contain only 1 identifer
!type_binding
type_binding
: name_list ':' expression
```

```
    {
  type_binding.r_actual_parms=name_list.r_actual_parms;
  type_binding.fm_call_specs=
     name_list.fm_call_specs;
  name_list.update=type_binding.update;
  type_binding.fm_call_actuals=
     name_list.fm_call_actuals;

  name_list.remove=type_binding.remove;
  type_binding.r_call_actuals=
     name_list.r_call_actuals;
  type_binding.r_call_specs=
     name_list.r_call_specs;
  type_binding.r_parm_count=name_list.r_parm_count;

type_binding.mpkg_gets=name_list.mpkg_gets;
type_binding.g_parm_decl=["\t",name_list.identifier_list,": ",
            expression.type_mark,"\n"];
type_binding.fm_parm_spec=
   ["\t\t\t\t",name_list.identifier_list,": ",
            expression.type_mark,"\n"];
type_binding.r_parm_spec=
   ["\t\t\t\t",name_list.identifier_list,": ",
            expression.type_mark,"\n"];

type_binding.c_parm_spec=
   [name_list.identifier_list,": ",expression.type_mark];
type_binding.q_parm_specs=
   [name_list.identifier_list,": ",expression.type_mark];
type_binding.cr_type_mark=expression.type_mark;
type_binding.c_parm_decl=
   ["\t\t",name_list.identifier_list,": ",
            expression.type_mark,"\n"];
type_binding.cr_parm=name_list.identifier_list;
type_binding.r_parm=name_list.identifier_list;


type_binding.RE_fm_actual_parm=
   ["\t\t\t\t\t\t\t",name_list.identifier_list,"\n"];
type_binding.RE_r_actual_parm=
   ["\t\t\t\t\t\t\tR_Parm,\n"];
type_binding.r_type_mark=expression.type_mark;
type_binding.g_actual_parms=name_list.identifier_list;

name_list.type_mark=expression.type_mark;
type_binding.gen_loop_vars=name_list.gen_loop_vars;
type_binding.rpkg_puts=name_list.rpkg_puts;
type_binding.text=[name_list.text," : ",expression.text];
   }
| '$' NAME ':' expression
   {
```

```
    }
;
!name_list
name_list
: name_list NAME
    {
  name_list[1].r_actual_parms=
    [name_list[2].r_actual_parms,", ",NAME.%text];
  name_list[2].update=name_list[1].update;
  name_list[1].fm_call_actuals=
     (name_list[1].update(NAME.%text) == "false")
     > [name_list[2].fm_call_actuals," , ",
          NAME.%text]
     #  [name_list[2].fm_call_actuals," , ",
          name_list[1].update(NAME.%text)];

  name_list[1].fm_call_specs=
     (name_list[1].update(NAME.%text) == "false")
     > [name_list[2].fm_call_specs,";\n",
          NAME.%text," : ", name_list[1].type_mark]
     #  [name_list[2].fm_call_specs," ;\n",
          NAME.%text,": in out ", name_list[1].type_mark];

  name_list[2].remove=name_list[1].remove;

  ! "remove" is a map which indicates that a variable should be
  ! omitted from the list of actuals.  As a result,
  ! name_list[2].actuals may be empty.  This conditional checks
  ! empty cases in order to properly place the delimiter ",".
  name_list[1].r_call_actuals=
     (name_list[1].remove(NAME.%text) == "true")
     > name_list[2].r_call_actuals
     #  (name_list[2].r_call_actuals == "")
          > NAME.%text
          # [name_list[2].r_call_actuals," , ",NAME.%text];

  ! see name_list[2].r_call_actuals comment above.
  name_list[1].r_call_specs=
     (name_list[1].remove(NAME.%text) == "true")
     > name_list[2].r_call_specs
     #  (name_list[2].r_call_specs == "")
          > [NAME.%text,": out ",name_list[1].type_mark]
          #  [name_list[2].r_call_specs," ;\n",
               NAME.%text,": out ", name_list[1].type_mark];

  name_list[1].r_parm_count=name_list[2].r_parm_count + 1;

name_list.mpkg_gets=
```

```
    [name_list.mpkg_gets,
      "\t\t\tIMPLEMENTATION.GET(INFILE,",NAME.%text,");\n"];
name_list[1].identifier_list=
    [name_list[2].identifier_list,", ",NAME.%text];
    name_list[1].gen_loop_vars=
[name_list[2].gen_loop_vars,";",
  NAME.%text,":",name_list[1].type_mark];

name_list[2].type_mark=name_list[1].type_mark;
name_list[1].rpkg_puts=[name_list[2].rpkg_puts,
    "\t\tPUT(outfile,\"",NAME.%text," = \");\n",
    "\t\t\tIMPLEMENTATION.PUT(outfile,",NAME.%text,");\n",
"\t\tNEW_LINE(outfile);\n"];
name_list[1].text=[name_list[2].text," ",NAME.%text];
    }
| NAME
    {
name_list.r_actual_parms=NAME.%text;
name_list.fm_call_actuals=
    (name_list[1].update(NAME.%text)=="false")
    > NAME.%text
    #  name_list.update(NAME.%text);

name_list.fm_call_specs=
    (name_list.update(NAME.%text)=="false")
    > [NAME.%text,": ",name_list.type_mark]
    #  [NAME.%text,": in out ",name_list.type_mark];

name_list[1].r_call_actuals=
    (name_list.remove(NAME.%text) == "true")
    > ""
    #  NAME.%text;

name_list.r_call_specs=
    (name_list.remove(NAME.%text)=="true")
    > ""
    #  [NAME.%text,": out ",name_list.type_mark];

name_list.r_parm_count=1;

name_list.mpkg_gets=
["\t\t\tIMPLEMENTATION.GET(INFILE,",NAME.%text,");\n"];
name_list.identifier_list=[NAME.%text];
name_list.gen_loop_vars=[NAME.%text,":",name_list.type_mark];
name_list.rpkg_puts=
["\t\tPUT(outfile,\"",NAME.%text," =\");\n",
  "\t\tPUT(outfile,",NAME.%text,");\n",
  "\t\tNEW_LINE(outfile);\n"];
name_list.text=NAME.%text;
    }
;
```

```
!restriction
restriction
: SUCH expression_list
   {
restriction.such_quantifier_trans=
   expression_list.when_expr_list_trans;
restriction.text=[" ",SUCH.%text," ",expression_list.text];

!return parameter should never be part of restrictions
expression_list.r_parm="see restriction production";
   }
|
   {
!when no restrictions exist on the quantifier,
!the checking test will always be conducted.
restriction.such_quantifier_trans=
   " (true) ";
restriction.text="";

   }
;
!optional_actual_name
optional_actual_name
: actual_name
   {
optional_actual_name.text=actual_name.text;


   }
|
   {
optional_actual_name.text="";


   }
;
!actual_name
actual_name
: NAME '{' actuals '}'
   {
actual_name.text=[NAME.%text,"{",actuals.text,"}"];


   }
| NAME%prec SEMI! must have a lower precedence than '{'
   {
actual_name.type_mark=NAME.%text;
actual_name.identifier=NAME.%text;
  actual_name.text=NAME.%text;
```

```
    }
;
!actuals
actuals
: actuals ',' arg%prec COMMA
    {
!actuals[2] will contain only one variable
actuals[1].init_statements=
    ["\t\t\t",arg.text," := ",actuals[2].text,";\n"];
actuals[1].remove_parm=arg.text;
actuals[1].update={(actuals[2].text:arg.text)};
actuals[1].actual_parms=
    [actuals[2].actual_parms," , ",arg.actual_parm];
actuals[2].r_parm=actuals[1].r_parm;
arg.r_parm=actuals[1].r_parm;
  actuals[1].text=[actuals[2].text,",",arg.text];



    }
| arg
    {
actuals.actual_parms=[arg.actual_parm];
arg.r_parm=actuals.r_parm;
  actuals.text=arg.text;



    }
;
!arg
arg
: expression
    {
arg.actual_parm=expression.expr_trans;
expression.r_parm=arg.r_parm;
arg.text=expression.text;



    }
| pair
    {



    }
;
!expression_list
expression_list
```

```
: expression_list ',' expression%prec COMMA
   (
expression_list[1].q_with_clauses=
   [expression_list[2].q_with_clauses,
     expression.q_with_clauses];
expression_list[1].when_expr_list_trans=
[expression_list[2].when_expr_list_trans,
" and then\n\t\t\t", expression.expr_trans];

!where expression translation:  conditional translation
!requires unique translation generated at the conditional
! expression
expression_list[1].where_expr_list_trans=
   (expression.class == "conditional")
   >
[expression_list[2].where_expr_list_trans,
expression.conditional_trans]
   #
[expression_list[2].where_expr_list_trans,
"\t\t\t\tif not (",expression.expr_trans,") then\n",
"\t\t\t\t\tREPORT.ERROR(condition,\n",
expression_list[1].RE_actual_parms,
"\t\t\t\t\t\"",expression_list[1].in_err_msg_when,
" NOT ",expression.text,"\");\n",
"\t\t\t\tend if;\n"];

!pass error message and parameters for conditional translation
expression.err_msg=expression_list[1].in_err_msg_when;
expression.RE_actual_parms=expression_list[1].RE_actual_parms;

!pass information necessary to determine how to translate
!the <=> and = expressions.
expression_list[2].cr_parm=expression_list[1].cr_parm;
expression.cr_parm=expression_list[1].cr_parm;
expression.is_leftmost="false";

expression_list[2].r_parm=expression_list[1].r_parm;
expression.r_parm=expression_list[1].r_parm;
expression_list.out_err_msg_when=
["WHEN ",expression_list[2].text];
  expression_list[2].in_err_msg_when=
expression_list[1].in_err_msg_when;
expression_list[1].text=
[expression_list[2].text,",",expression.text];
expression_list[2].RE_actual_parms=
   expression_list[1].RE_actual_parms;

expression_list[1].quantifier_functions=
   [expression_list[2].quantifier_functions,
     expression.quantifier_functions];
```

```
expression_list[1].q_iterator_macros=
   [expression_list[2].q_iterator_macros,
    expression.q_iterator_macros];
   }
| expression%prec COMMA
   {
expression_list.q_with_clauses=expression.q_with_clauses;
expression_list.when_expr_list_trans=
[expression.expr_trans];

!where expression translation, conditional translation
!requires unique translation generated at the conditional
! expression
expression_list.where_expr_list_trans=
   (expression.class == "conditional")
   >
         expression.conditional_trans
   #
         ["\t\t\t\tif not (",
expression.expr_trans,") then\n",
"\t\t\t\t\tREPORT.ERROR(condition,\n",
expression_list[1].RE_actual_parms,
"\t\t\t\t\t\"",expression_list.in_err_msg_when,
" NOT ",expression.text,"\");\n",
"\t\t\t\tend if;\n"];

!pass error message and parameters for conditional translation
expression.err_msg=expression_list.in_err_msg_when;
expression.RE_actual_parms=expression_list.RE_actual_parms;

expression.cr_parm=expression_list.cr_parm;
expression.is_leftmost="true";
expression_list.c_where_expr_list_trans=
[expression.expr_trans];
expression.r_parm=expression_list.r_parm;
expression_list.out_err_msg_when=
["WHEN ",expression.text];
expression_list[1].text=expression.text;

expression_list.quantifier_functions=
   expression.quantifier_functions;

expression_list.q_iterator_macros=
   expression.q_iterator_macros;
   }
;
!expression
expression
: QUANTIFIER '(' formals BIND expression ')'
   {
!quantifier
```

```
expression[1].q_with_clauses=
 ["with GEN_",QUANTIFIER.%text,"_",i2s(QUANTIFIER.%line),";\n",
    expression[2].q_with_clauses];

expression.class="quantifier";

! quantifier function
expression[1].quantifier_functions=
   (QUANTIFIER.%text == "ALL")
   >["\t\t",formals.q_parm_specs,";\n",
        "\t\tfunction ALL_",i2s(QUANTIFIER.%line),
      " return boolean is\n",
          "\t\t\tvalue:  boolean := true;\n",
          expression[2].quantifier_functions,
        "\t\tbegin\n",
          "\t\t\tforeach([",formals.gen_loop_vars,"],",
   "GEN_ALL_",i2s(QUANTIFIER.%line),",[assurance],[\n",
              "\t\t\t\tif (value = true) then\n",
    QUANTIFIER.all_checking_trans,
          "\t\t\tend if;])\n",
          "\t\t\treturn value;\n",
        "\t\tend ALL_",i2s(QUANTIFIER.%line),";\n"]
    # (QUANTIFIER.%text == "SOME")
       >["\t\t",formals.q_parm_specs,";\n",
            "\t\tfunction SOME_",i2s(QUANTIFIER.%line),
         " return boolean is\n",
              "\t\t\tvalue:  boolean := false;\n",
                expression[2].quantifier_functions,
            "\t\tbegin\n",
              "\t\t\tforeach([",formals.gen_loop_vars,"],",
   "GEN_SOME_",i2s(QUANTIFIER.%line),",[assurance],[\n",
                "\t\t\t\tif (value = false) then\n",
       QUANTIFIER.some_checking_trans,
    "\t\t\tend if;])\n",
              "\t\t\treturn value;\n",
          "\t\tend SOME_",i2s(QUANTIFIER.%line),";\n"]
       # "";

! all checking logic
!logic searches for a single false case for failure.
QUANTIFIER.all_checking_trans=
   ["\t\t\t\tif ",formals.such_quantifier_trans," then\n",
        "\t\t\t\t\tif not ",expression[2].expr_trans," then\n",
          "\t\t\t\t\t\tvalue := false;\n",
        "\t\t\t\t\tend if;\n",
    "\t\t\tend if;\n"];

! some checking logic
!logic searches for a single true case for satisfaction.
QUANTIFIER.some_checking_trans=
   ["\t\t\t\tif ",formals.such_quantifier_trans," then\n",
```

```
        "\t\t\t\t\t\tif ",expression[2].expr_trans," then\n",
            "\t\t\t\t\t\t\tvalue := true;\n",
        "\t\t\t\t\t\tend if;\n",
    "\t\t\t\tend if;\n"];

expression[2].r_parm=expression[1].r_parm;

! expression translation
expression[1].expr_trans=
   (QUANTIFIER.%text == "ALL")
   > ["ALL_",i2s(QUANTIFIER.%line)]
   #  (QUANTIFIER.%text == "SOME")
   > ["SOME_",i2s(QUANTIFIER.%line)]
   #  "undefined quantifier";

! expression text
expression[1].text= [QUANTIFIER.%text,"(",formals.text," ",
   BIND.%text," ",expression[2].text,")"];

expression[2].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;

! iterator macros
          expression[1].q_iterator_macros=
 [expression[2].q_iterator_macros,"\n",
    "--**put with and use statements here**--\n",
    "generator(GEN_",QUANTIFIER.%text,"_",i2s(QUANTIFIER.%line),
    ",\[assurance:  float\], ",
    "\[",formals.gen_loop_vars,"\],\n",
    "\[is\n",
        "\t--**put any required declarations here**--\n",
    "begin\n",
        "\t--**put iterating statements here**--\n",
        "\tgenerate(--**put generated values here**);\n",
        "\t--**put more statements here as required**--\n",
    "end GEN_",QUANTIFIER.%text,"_",i2s(QUANTIFIER.%line),";\]\n"];

   }
| actual_name ! variables and constants
   {
!actual_name
!n
  expression.text=actual_name.identifier;
expression.expr_trans= actual_name.identifier;
expression.type_mark=actual_name.type_mark;

expression.class="no_special_handling";
expression.quantifier_functions="";
expression.q_with_clauses="";
expression.q_iterator_macros="";
   }
```

```
| expression '(' actuals ')'! function call
   {
!function call
!n
expression[1].q_with_clauses="";
expression[1].expr_trans=
   [expression[2].expr_trans,"(",actuals.actual_parms,")"];
expression[2].r_parm=expression[1].r_parm;
actuals.r_parm=expression[1].r_parm;
expression[1].text=[expression[2].text,"(",actuals.text,")"];

expression.class="no_special_handling";
expression.quantifier_functions="";
expression.q_iterator_macros="";
   }
| expression '@' actual_name! expression with explicit type cast
   {
!type cast
!n
!assumes overloaded enumeration type handled by ADA as
! actual_name.text'(expr)
expression.expr_trans=
[actual_name.text,"'(",expression[2].expr_trans,")"];
expression.text=[expression[2].text,"@",actual_name.text];

expression[1].q_with_clauses="";
expression.class="no_special_handling";
expression[1].quantifier_functions="";
expression.q_iterator_macros="";
   }
| NOT expression%prec NOT
   {
!not
!n
expression[1].q_with_clauses=expression[2].q_with_clauses;
expression[1].expr_trans=
   [" not (",expression[2].expr_trans,")"];
expression[2].r_parm=expression[1].r_parm;
expression[1].text=[NOT.%text,expression[2].text];
expression.class="no_special_handling";
expression[1].quantifier_functions=
   expression[2].quantifier_functions;
expression[1].q_iterator_macros=
   expression[2].q_iterator_macros;
expression[2].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
   }
| expression AND expression%prec AND
   {
!and
!n
```

```
expression[1].q_with_clauses=
   [expression[2].q_with_clauses,
     expression[3].q_with_clauses];

expression[1].expr_trans=
   ["(",expression[2].expr_trans," and ",
          expression[3].expr_trans,")"];
expression[2].r_parm=expression[1].r_parm;
expression[3].r_parm=expression[1].r_parm;

expression[1].text=
   [expression[2].text,AND.%text,expression[3].text];

expression.class="no_special_handling";

expression[1].quantifier_functions=
   [expression[2].quantifier_functions,
     expression[3].quantifier_functions];

expression[1].q_iterator_macros=
   [expression[2].q_iterator_macros,
     expression[3].q_iterator_macros];

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;

   }
| expression OR expression%prec OR
   {
!or
!n
expression[1].q_with_clauses=
   [expression[2].q_with_clauses,
     expression[3].q_with_clauses];

expression[1].expr_trans=
   ["(",expression[2].expr_trans,
      " or ",expression[3].expr_trans,")"];
expression[2].r_parm=expression[1].r_parm;
expression[3].r_parm=expression[1].r_parm;

expression[1].text=
   [expression[2].text,OR.%text,expression[3].text];


expression.class="no_special_handling";
```

```
expression[1].quantifier_functions=
   [expression[2].quantifier_functions,
     expression[3].quantifier_functions];

expression[1].q_iterator_macros=
   [expression[2].q_iterator_macros,
     expression[3].q_iterator_macros];
!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;

   }
| expression IMPLIES expression%prec IMPLIES
   {
!implies
!n
expression[1].q_with_clauses=
   [expression[2].q_with_clauses,
     expression[3].q_with_clauses];

expression[2].r_parm=expression[1].r_parm;
expression[3].r_parm=expression[1].r_parm;

!implemented as a function call, where "implies" <=> "=>"
expression[1].expr_trans=
   ["implies(",expression[2].expr_trans,",",
      expression[3].expr_trans,")"];
expression[1].text=
   [expression[2].text,IMPLIES.%text,expression[3].text];


expression[1].quantifier_functions=
   [expression[2].quantifier_functions,
     expression[3].quantifier_functions];

expression[1].q_iterator_macros=
   [expression[2].q_iterator_macros,
     expression[3].q_iterator_macros];
expression.class="no_special_handling";

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;
   }
| expression IFF expression%prec IFF
```

```
   {
!iff
!n
expression[1].q_with_clauses=
   [expression[2].q_with_clauses,
     expression[3].q_with_clauses];

            !If this is the leftmost expression and expression[2]
!is the concept return parameter, then generate code to
!return the value computed by the translation of expression[3].
!iff implemented as function call, where
!"iff(x,y)" <=> "x <=> y"
expression[1].expr_trans=
   (expression[2].expr_trans==expression[1].cr_parm &&
     expression[1].is_leftmost=="true")
>["\t\treturn",expression[3].expr_trans,";\n"]
# ["iff(",expression[2].expr_trans,",",
     expression[3].expr_trans,")"];

expression[1].text=
   [expression[2].text,IFF.%text,expression[3].text];

expression.class="no_special_handling";

expression[1].quantifier_functions=
   [expression[2].quantifier_functions,
     expression[3].quantifier_functions];

expression[1].q_iterator_macros=
   [expression[2].q_iterator_macros,
     expression[3].q_iterator_macros];

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;

expression[2].r_parm=expression[1].r_parm;
expression[3].r_parm=expression[1].r_parm;

   }
| expression '=' expression%prec LE
   {
!=
!n
expression[1].q_with_clauses=
   [expression[2].q_with_clauses,
     expression[3].q_with_clauses];
```

```
            !see comments at IFF expression
expression[1].expr_trans=
   (expression[1].is_leftmost=="true" &&
     expression[2].expr_trans==expression[1].cr_parm)
> ["\t\treturn",expression[3].expr_trans,";\n"]
#  ["(",expression[2].expr_trans,
     " = ",expression[3].expr_trans,")"];


expression[1].text=
   [expression[2].text," = ",expression[3].text];

expression.class="no_special_handling";

expression[1].quantifier_functions=
   [expression[2].quantifier_functions,
     expression[3].quantifier_functions];

expression[1].q_iterator_macros=
   [expression[2].q_iterator_macros,
     expression[3].q_iterator_macros];

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;

expression[2].r_parm=expression[1].r_parm;
expression[3].r_parm=expression[1].r_parm;

   }
| expression '<' expression%prec LE
   {
!<
!n
expression[1].q_with_clauses=
   [expression[2].q_with_clauses,
     expression[3].q_with_clauses];

expression[1].expr_trans=
   ["(",expression[2].expr_trans,
     " < ",expression[3].expr_trans,")"];
expression[2].r_parm=expression[1].r_parm;
expression[3].r_parm=expression[1].r_parm;

expression[1].text=[expression[2].text,"<",expression[3].text];

expression.class="no_special_handling";
```

```
expression[1].quantifier_functions=
   [expression[2].quantifier_functions,
     expression[3].quantifier_functions];

expression[1].q_iterator_macros=
   [expression[2].q_iterator_macros,
     expression[3].q_iterator_macros];

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;

   }
| expression '>' expression%prec LE
   {
!>
!n
expression[1].q_with_clauses=
   [expression[2].q_with_clauses,
     expression[3].q_with_clauses];

expression[1].expr_trans=
   ["(",expression[2].expr_trans,
      " > ",expression[3].expr_trans,")"];
expression[2].r_parm=expression[1].r_parm;
expression[3].r_parm=expression[1].r_parm;

expression[1].text=
   [expression[2].text,">",expression[3].text];

expression.class="no_special_handling";

expression[1].quantifier_functions=
   [expression[2].quantifier_functions,
     expression[3].quantifier_functions];

expression[1].q_iterator_macros=
   [expression[2].q_iterator_macros,
     expression[3].q_iterator_macros];

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;

   }
```

```
| expression LE expression%prec LE
   {
!<=
!n
expression[1].q_with_clauses=
   [expression[2].q_with_clauses,
     expression[3].q_with_clauses];

expression[1].expr_trans=
   ["(",expression[2].expr_trans,
      " <= ",expression[3].expr_trans,")"];
expression[2].r_parm=expression[1].r_parm;
expression[3].r_parm=expression[1].r_parm;

expression[1].text=
   [expression[2].text,LE.%text,expression[3].text];

expression.class="no_special_handling";

expression[1].quantifier_functions=
   [expression[2].quantifier_functions,
     expression[3].quantifier_functions];

expression[1].q_iterator_macros=
   [expression[2].q_iterator_macros,
     expression[3].q_iterator_macros];

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;

   }
| expression GE expression%prec LE
   {
!>=
!n
expression[1].q_with_clauses=
   [expression[2].q_with_clauses,
     expression[3].q_with_clauses];

expression[1].expr_trans=
   ["(",expression[2].expr_trans,
      " >= ",expression[3].expr_trans,")"];
expression[2].r_parm=expression[1].r_parm;
expression[3].r_parm=expression[1].r_parm;

expression[1].text=
   [expression[2].text,GE.%text,expression[3].text];
```

```
expression.class="no_special_handling";

expression[1].quantifier_functions=
   [expression[2].quantifier_functions,
     expression[3].quantifier_functions];

expression[1].q_iterator_macros=
   [expression[2].q_iterator_macros,
     expression[3].q_iterator_macros];

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;


   }
| expression NE expression%prec LE
   {
!~=
!n
expression[1].q_with_clauses=
   [expression[2].q_with_clauses,
     expression[3].q_with_clauses];

expression[1].expr_trans=
   ["(",expression[2].expr_trans,
      "/= ",expression[3].expr_trans,")"];
expression[2].r_parm=expression[1].r_parm;
expression[3].r_parm=expression[1].r_parm;

expression[1].text=
   [expression[2].text,NE.%text,expression[3].text];

expression.class="no_special_handling";

expression[1].quantifier_functions=
   [expression[2].quantifier_functions,
     expression[3].quantifier_functions];

expression[1].q_iterator_macros=
   [expression[2].q_iterator_macros,
     expression[3].q_iterator_macros];

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
```

```
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;

   }
| expression NLT expression%prec LE
   {
!~<
!n
expression[1].q_with_clauses=
   [expression[2].q_with_clauses,
     expression[3].q_with_clauses];

expression[1].expr_trans=
   ["(not(",expression[2].expr_trans,
      " < ",expression[3].expr_trans,"))"];
expression[2].r_parm=expression[1].r_parm;
expression[3].r_parm=expression[1].r_parm;

expression[1].text=
   [expression[2].text,NLT.%text,expression[3].text];

expression.class="no_special_handling";

expression[1].quantifier_functions=
   [expression[2].quantifier_functions,
     expression[3].quantifier_functions];

expression[1].q_iterator_macros=
   [expression[2].q_iterator_macros,
     expression[3].q_iterator_macros];

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;

   }
| expression NGT expression%prec LE
   {
!~>
!n
expression[1].q_with_clauses=
   [expression[2].q_with_clauses,
     expression[3].q_with_clauses];

expression[1].expr_trans=
   ["(not(",expression[2].expr_trans,
      " > ",expression[3].expr_trans,"))"];
expression[2].r_parm=expression[1].r_parm;
```

```
expression[3].r_parm=expression[1].r_parm;

expression[1].text=
   [expression[2].text,NGT.%text,expression[3].text];

expression.class="no_special_handling";

expression[1].quantifier_functions=
   [expression[2].quantifier_functions,
     expression[3].quantifier_functions];

expression[1].q_iterator_macros=
   [expression[2].q_iterator_macros,
     expression[3].q_iterator_macros];

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;

   }
| expression NLE expression%prec LE
   {
!~<=
!n
expression[1].q_with_clauses=
   [expression[2].q_with_clauses,
     expression[3].q_with_clauses];

          expression[1].expr_trans=
 ["(not(",expression[2].expr_trans,
      " <= ",expression[3].expr_trans,"))"];
expression[2].r_parm=expression[1].r_parm;
expression[3].r_parm=expression[1].r_parm;

expression[1].text=
   [expression[2].text,NLE.%text,expression[3].text];

expression.class="no_special_handling";

expression[1].quantifier_functions=
   [expression[2].quantifier_functions,
     expression[3].quantifier_functions];

expression[1].q_iterator_macros=
   [expression[2].q_iterator_macros,
     expression[3].q_iterator_macros];

!info required to determine when we have reached the leftmost
```

```
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;

   }
| expression NGE expression%prec LE
   {
!~>=
!n
expression[1].q_with_clauses=
   [expression[2].q_with_clauses,
     expression[3].q_with_clauses];

            expression[1].expr_trans=
  ["(not(",expression[2].expr_trans,
     " >= ",expression[3].expr_trans,"))"];
expression[2].r_parm=expression[1].r_parm;
expression[3].r_parm=expression[1].r_parm;

expression[1].text=
   [expression[2].text,NGE.%text,expression[3].text];

expression.class="no_special_handling";

expression[1].quantifier_functions=
   [expression[2].quantifier_functions,
     expression[3].quantifier_functions];

expression[1].q_iterator_macros=
   [expression[2].q_iterator_macros,
     expression[3].q_iterator_macros];

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;

   }
| expression EQV expression%prec LE
   {
!==
!n
expression[1].q_with_clauses=
   [expression[2].q_with_clauses,
     expression[3].q_with_clauses];

expression[1].expr_trans=
```

```
    ["equivalent(",expression[2].expr_trans," , ",
       expression[3].expr_trans,")"];
expression[2].r_parm=expression[1].r_parm;
expression[3].r_parm=expression[1].r_parm;
 expression[1].text=
   [expression[2].text,EQV.%text,expression[3].text];

expression.class="no_special_handling";

expression[1].quantifier_functions=
   [expression[2].quantifier_functions,
     expression[3].quantifier_functions];

expression[1].q_iterator_macros=
   [expression[2].q_iterator_macros,
     expression[3].q_iterator_macros];

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;

   }
| expression NEQV expression%prec LE
   {
!~==
!n
expression[1].q_with_clauses=
   [expression[2].q_with_clauses,
     expression[3].q_with_clauses];

          expression[1].expr_trans=
 ["NOT equivalent(",expression[2].expr_trans,
      " , ",expression[3].expr_trans,")"];
expression[2].r_parm=expression[1].r_parm;
expression[3].r_parm=expression[1].r_parm;
 expression[1].text=
   [expression[2].text,NEQV.%text,expression[3].text];

expression.class="no_special_handling";

expression[1].quantifier_functions=
   [expression[2].quantifier_functions,
     expression[3].quantifier_functions];

expression[1].q_iterator_macros=
   [expression[2].q_iterator_macros,
     expression[3].q_iterator_macros];
```

```
!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;

   }
| '-' expression%prec UMINUS
   {
!u-
!n
expression[1].q_with_clauses=expression[2].q_with_clauses;
expression[1].expr_trans=
   ["(-(",expression[2].expr_trans,"))"];
expression[2].r_parm=expression[1].r_parm;
 expression[1].text=["-",expression[2].text];

expression.class="no_special_handling";

expression[1].quantifier_functions=
   expression[2].quantifier_functions;

expression[1].q_iterator_macros=
   expression[2].q_iterator_macros;

expression[2].is_leftmost="true";
expression[2].cr_parm=expression[1].cr_parm;
   }
| expression '+' expression%prec PLUS
   {
!+
!n
expression[1].q_with_clauses=
   [expression[2].q_with_clauses,
     expression[3].q_with_clauses];

expression[1].expr_trans=
   ["(",expression[2].expr_trans,
      " + ",expression[3].expr_trans,")"];
expression[2].r_parm=expression[1].r_parm;
expression[3].r_parm=expression[1].r_parm;
 expression[1].text=
   [expression[2].text,"+",expression[3].text];

expression.class="no_special_handling";

expression[1].quantifier_functions=
   [expression[2].quantifier_functions,
     expression[3].quantifier_functions];
```

```
expression[1].q_iterator_macros=
   [expression[2].q_iterator_macros,
     expression[3].q_iterator_macros];

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;

   }
| expression '-' expression%prec MINUS
   {
!-
!n
expression[1].q_with_clauses=
   [expression[2].q_with_clauses,
     expression[3].q_with_clauses];

expression[1].expr_trans=
   ["(",expression[2].expr_trans,
      " - ",expression[3].expr_trans,")"];
expression[2].r_parm=expression[1].r_parm;
expression[3].r_parm=expression[1].r_parm;
 expression[1].text=
   [expression[2].text,"-",expression[3].text];

expression.class="no_special_handling";

expression[1].quantifier_functions=
   [expression[2].quantifier_functions,
     expression[3].quantifier_functions];

expression[1].q_iterator_macros=
   [expression[2].q_iterator_macros,
     expression[3].q_iterator_macros];

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;

   }
| expression '*' expression%prec MUL
   {
!*
!n
expression[1].q_with_clauses=
```

```
    [expression[2].q_with_clauses,
      expression[3].q_with_clauses];

expression[1].expr_trans=
    ["(",expression[2].expr_trans,
        " * ",expression[3].expr_trans,")"];
expression[2].r_parm=expression[1].r_parm;
expression[3].r_parm=expression[1].r_parm;
  expression[1].text=
    [expression[2].text," * ",expression[3].text];

expression.class="no_special_handling";
expression[1].quantifier_functions=
    [expression[2].quantifier_functions,
      expression[3].quantifier_functions];

expression[1].q_iterator_macros=
    [expression[2].q_iterator_macros,
      expression[3].q_iterator_macros];

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;


    }
| expression '/' expression%prec DIV
    {
!/
!n
expression[1].q_with_clauses=
    [expression[2].q_with_clauses,
      expression[3].q_with_clauses];

expression[1].expr_trans=
    ["(",expression[2].expr_trans,
        " / ",expression[3].expr_trans,")"];
expression[2].r_parm=expression[1].r_parm;
expression[3].r_parm=expression[1].r_parm;
  expression[1].text=
    [expression[2].text," / ",expression[3].text];

expression.class="no_special_handling";

expression[1].quantifier_functions=
    [expression[2].quantifier_functions,
      expression[3].quantifier_functions];
```

```
expression[1].q_iterator_macros=
   [expression[2].q_iterator_macros,
     expression[3].q_iterator_macros];

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;


   }
| expression MOD expression%prec MOD
   {
!mod
!n
expression[1].q_with_clauses=
   [expression[2].q_with_clauses,
     expression[3].q_with_clauses];

expression[1].expr_trans=
   ["(",expression[2].expr_trans,
      " mod ",expression[3].expr_trans,")"];
expression[2].r_parm=expression[1].r_parm;
expression[3].r_parm=expression[1].r_parm;
  expression[1].text=
   [expression[2].text," ",MOD.%text," ",expression[3].text];

expression.class="no_special_handling";

expression[1].quantifier_functions=
   [expression[2].quantifier_functions,
     expression[3].quantifier_functions];

expression[1].q_iterator_macros=
   [expression[2].q_iterator_macros,
     expression[3].q_iterator_macros];

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;

   }
| expression EXP expression%prec EXP
   {
!**
!n
```

```
expression[1].q_with_clauses=
   [expression[2].q_with_clauses,
     expression[3].q_with_clauses];

expression[1].expr_trans=
   ["(",expression[2].expr_trans,
      " ** ",expression[3].expr_trans,")"];
expression[2].r_parm=expression[1].r_parm;
expression[3].r_parm=expression[1].r_parm;
 expression[1].text=
   [expression[2].text,$2.%text,expression[3].text];

expression.class="no_special_handling";

expression[1].quantifier_functions=
   [expression[2].quantifier_functions,
     expression[3].quantifier_functions];

expression[1].q_iterator_macros=
   [expression[2].q_iterator_macros,
     expression[3].q_iterator_macros];

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;

   }
| expression U expression%prec U
   {
!U
!n
expression[1].q_with_clauses=
   [expression[2].q_with_clauses,
     expression[3].q_with_clauses];

expression[1].expr_trans=
   ["union(",expression[2].expr_trans,
      " , ",expression[3].expr_trans,")"];
expression[2].r_parm=expression[1].r_parm;
expression[3].r_parm=expression[1].r_parm;
 expression[1].text=
   [expression[2].text," ",U.%text," ",expression[3].text];

expression.class="no_special_handling";

expression[1].quantifier_functions=
   [expression[2].quantifier_functions,
     expression[3].quantifier_functions];
```

```
expression[1].q_iterator_macros=
   [expression[2].q_iterator_macros,
     expression[3].q_iterator_macros];

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;

   }
| expression APPEND expression%prec APPEND
   {
!append
!n
expression[1].q_with_clauses=
   [expression[2].q_with_clauses,
     expression[3].q_with_clauses];

!translated as ADA concatenation operator
expression[1].expr_trans=
   ["(",expression[2].expr_trans,
      " & ",expression[3].expr_trans,")"];

expression[1].text=
   [expression[2].text," ",APPEND.%text," ",
      expression[3].text];

expression[2].r_parm=expression[1].r_parm;
expression[3].r_parm=expression[1].r_parm;


expression.class="no_special_handling";

expression[1].quantifier_functions=
   [expression[2].quantifier_functions,
     expression[3].quantifier_functions];

expression[1].q_iterator_macros=
   [expression[2].q_iterator_macros,
     expression[3].q_iterator_macros];

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;
```

```
    }
| expression IN expression%prec IN
    {
!in
!n
expression[1].q_with_clauses=
    [expression[2].q_with_clauses,
      expression[3].q_with_clauses];


expression[1].expr_trans=
    ["(",expression[2].expr_trans," IN ",
        expression[3].expr_trans,")"];

  expression[1].text=
    [expression[2].text," ",IN.%text," ",expression[3].text];

expression[2].r_parm=expression[1].r_parm;
expression[3].r_parm=expression[1].r_parm;

expression[1].class="no_special_handling";

expression[1].quantifier_functions=
    [expression[2].quantifier_functions,
      expression[3].quantifier_functions];

expression[1].q_iterator_macros=
    [expression[2].q_iterator_macros,
      expression[3].q_iterator_macros];

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;

    }
| '*' expression %prec STAR
    ! *x is the value of x in the previous state
    {
!*e
!n

expression[1].q_with_clauses="";
  expression[1].text=["*",expression[2].text];
expression.class="no_special_handling";
expression[1].quantifier_functions="";
expression[1].q_iterator_macros="";
    }
| '$' expression%prec DOT
```

```
    ! $x represents a collection of items rather than just one
    ! s1 = {x, $s2} means s1 = union({x}, s2)
    ! s1 = [x, $s2] means s1 = append([x], s2)
    {
!$
!n
expression[1].q_with_clauses="";
  expression[1].text=["$",expression[2].text];
expression.class="no_special_handling";
expression[1].quantifier_functions="";
expression[1].q_iterator_macros="";

    }
| expression RANGE expression%prec RANGE
    ! x in [a .. b] iff x in {a .. b} iff a <= x <= b, [a .. b] is sorted in increasing order
    {
!range
!n
expression[1].q_with_clauses=
    [expression[2].q_with_clauses,
      expression[3].q_with_clauses];

expression[1].expr_trans=
    [expression[2].expr_trans,"..",expression[3].expr_trans];

  expression[1].text=
    [expression[2].text," ",RANGE.%text," ",expression[3].text];

expression.class="no_special_handling";

!e.g., quantifier in range [a..SUM( ... )] expression
expression[1].quantifier_functions=
    [expression[2].quantifier_functions,
      expression[3].quantifier_functions];

expression[1].q_iterator_macros=
    [expression[2].q_iterator_macros,
      expression[3].q_iterator_macros];

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;


    }
| expression '.' NAME%prec DOT
    {
!dot
```

```
!n
expression[1].q_with_clauses="";

!translated under assumption of implementation as record
expression[1].expr_trans=
   [expression[2].expr_trans,".",NAME.%text];

  expression[1].text=[expression[2].text,".",NAME.%text];
expression.class="no_special_handling";

expression[1].quantifier_functions="";
expression[1].q_iterator_macros="";

expression[2].is_leftmost="true";
expression[2].cr_parm=expression[1].cr_parm;
   }
| expression '[' expression ']'%prec DOT
   {
![]
!n
expression[1].q_with_clauses="";

!translation assumes implementation by ADA array
expression[1].expr_trans=
   [expression[2].expr_trans,"(",expression[2].expr_trans,")"];

  expression[1].text=
   [expression[2].text," [",expression[3].text,"] "];

expression.class="no_special_handling";

expression[1].quantifier_functions="";
expression[1].q_iterator_macros="";

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="true";
expression[3].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;

   }
| '(' expression ')'
   {
!()
!n
expression[1].q_with_clauses=[expression[2].q_with_clauses];

expression[1].expr_trans=["(",expression[2].expr_trans,")"];
  expression[1].text=["(",expression[2].text,")"];
    expression.class="no_special_handling";
```

```
expression[1].quantifier_functions=
   expression[2].quantifier_functions;
expression[1].q_iterator_macros=
   expression[2].q_iterator_macros;

expression[2].is_leftmost="true";
expression[2].cr_parm=expression[1].cr_parm;
   }
| '(' expression NAME ')'! expression with units of measurement
   ! standard time units: NANOSEC MICROSEC MILLISEC SECONDS MINUTES HOURS
   DAYS WEEKS
   {
!(TIME)
!n
expression[1].quantifier_functions="";

!there is nothing to support this translation
expression[1].expr_trans=
   ["(",expression[2].expr_trans," ",NAME.%text,")"];

expression[1].text=["(",expression[2].text,NAME.%text,")"];

expression.class="no_special_handling";
expression[1].q_iterator_macros="";

   }
| TIME! The current local time, used in temporal events
   {
!time
!n
expression[1].q_with_clauses="";

!call on system clock
expression[1].expr_trans="CLOCK";

 expression[1].text=TIME.%text;

expression.class="no_special_handling";
expression[1].quantifier_functions="";
expression[1].q_iterator_macros="";


   }
| DELAY! The time between the triggering event and the response
   {
!delay
!n
expression[1].q_with_clauses="";
 expression[1].text=DELAY.%text;
expression.class="no_special_handling";
expression[1].quantifier_functions="";
```

```
expression[1].q_iterator_macros="";

   }
| PERIOD! The time between successive events of this type
   {
!period
!n
expression[1].q_with_clauses="";
  expression[1].text=PERIOD.%text;
expression.class="no_special_handling";
expression[1].quantifier_functions="";
expression[1].q_iterator_macros="";

   }
| literal ! literal with optional type id
   {
!literal
!n
expression[1].q_with_clauses="";
expression[1].expr_trans=literal.identifier;
expression[1].text=literal.identifier;
expression.class="no_special_handling";
expression[1].quantifier_functions="";
expression[1].q_iterator_macros="";


   }
| '?'! An undefined value to be specified later
   {
!?
!n
expression[1].q_with_clauses="";
expression[1].text="?";
expression.class="no_special_handling";
expression[1].quantifier_functions="";
expression[1].q_iterator_macros="";
   }
| '!'! An undefined and illegal value
   {
!!
!n
expression[1].q_with_clauses="";
expression[1].text="!";
expression.class="no_special_handling";
expression[1].quantifier_functions="";
expression[1].q_iterator_macros="";

   }
| IF expression THEN expression middle_cases ELSE expression FI
   {
!conditional
```

```
!n
expression[1].q_with_clauses=
   [expression[2].q_with_clauses,
     expression[3].q_with_clauses,
     middle_cases.q_with_clauses,
     expression[4].q_with_clauses];

expression.conditional_trans=
   ["\t\t\tif (",expression[2].expr_trans,") then\n",
"\t\t\t\tif not (",expression[3].expr_trans,") then\n",
"\t\t\t\t\tREPORT.ERROR(condition,\n",
expression[1].RE_actual_parms,
"\t\t\t\t\t\"",expression[3].err_msg,"\");\n",
"\t\t\t\tend if;\n",
middle_cases.translation,
"\t\t\telse\n",
   "\t\t\t\tif not (",expression[4].expr_trans,") then\n",
"\t\t\t\t\tREPORT.ERROR(condition,\n",
expression[1].RE_actual_parms,
"\t\t\t\t\t\"",expression[4].err_msg,"\");\n",
"\t\t\t\tend if;\n",
"\t\t\tend if;\n"];

!error message when the antecedent is true but the
!consequence (THEN) is false
expression[3].err_msg=
   [expression[1].err_msg," IF ",expression[2].text," NOT ",
     expression[3].text];

!error message when the antecedent is false
!but the consequence (ELSE) is false
expression[4].err_msg=
   [expression[1].err_msg," IF-ELSE NOT ",expression[4].text];

!passes the appropriate part to the error message to
! middle_cases
middle_cases.err_msg=expression[1].err_msg;

!pass the error message parameters to middle_cases
middle_cases.RE_actual_parms=expression[1].RE_actual_parms;

!pass the return parameter name to appropriate children
middle_cases.r_parm=expression[1].r_parm;
expression[2].r_parm=expression[1].r_parm;
expression[3].r_parm=expression[1].r_parm;
expression[4].r_parm=expression[1].r_parm;

!a flag that indicates that this expression requires
!special handling.
expression[1].class="conditional";
```

```
!pass up all quantifier_functions
expression[1].quantifier_functions=
   [expression[2].quantifier_functions,
     expression[3].quantifier_functions,
     middle_cases.quantifier_functions,
     expression[4].quantifier_functions];

!pass up the quantifier iterator shells
expression[1].q_iterator_macros=
   [expression[2].q_iterator_macros,
     expression[3].q_iterator_macros,
     middle_cases.q_iterator_macros,
       expression[4].q_iterator_macros];

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[2].is_leftmost="false";
expression[3].is_leftmost="false";
expression[4].is_leftmost="false";
expression[2].cr_parm=expression[1].cr_parm;
expression[3].cr_parm=expression[1].cr_parm;
expression[4].cr_parm=expression[1].cr_parm;
middle_cases.cr_parm=expression[1].cr_parm;
   }
;
!middle_cases
middle_cases
: middle_cases ELSE_IF expression THEN expression
   {

middle_cases[1].q_with_clauses=
   [middle_cases[2].q_with_clauses,
     expression[1].q_with_clauses,
     expression[2].q_with_clauses];

middle_cases[1].translation=
   [middle_cases[2].translation,
     "\t\t\telsif",expression[1].expr_trans,"then\n",
     "\t\t\t\tif not (",expression[2].expr_trans,") then\n",
     "\t\t\t\t\tREPORT.ERROR(condition,\n",
     middle_cases[1].RE_actual_parms,
     "\t\t\t\t\t\"",expression[2].err_msg,"\");\n",
     "\t\t\t\tend if;\n"];

!error message when antecedent is true but consequence
!is false
expression[2].err_msg=
   [middle_cases[1].err_msg,
     " IF-ELSE_IF ",expression[1].text,
     " NOT ",expression[2].text];
```

```
!pass the appropriate portions of the error message
!to recursive middle_cases
middle_cases[2].err_msg=middle_cases[1].err_msg;

!pass the return parameter name to appropriate children
middle_cases[2].r_parm=middle_cases[1].r_parm;
expression[1].r_parm=middle_cases[1].r_parm;
expression[2].r_parm=middle_cases[1].r_parm;

!pass the actual parameters to recursive part.
middle_cases[2].RE_actual_parms=
   middle_cases[1].RE_actual_parms;

middle_cases[1].quantifier_functions=
   [middle_cases[2].quantifier_functions,
     expression[1].quantifier_functions,
     expression[2].quantifier_functions];

middle_cases[1].q_iterator_macros=
   [middle_cases[2].q_iterator_macros,
     expression[1].q_iterator_macros,
     expression[2].q_iterator_macros];

!info required to determine when we have reached the leftmost
!expression and, if so, how to translate it.
expression[1].is_leftmost="false";
expression[2].is_leftmost="false";
expression[1].cr_parm=middle_cases[1].cr_parm;
expression[2].cr_parm=middle_cases[1].cr_parm;
middle_cases[2].cr_parm=middle_cases[1].cr_parm;

   }
|
   {
middle_cases[1].q_with_clauses="";
middle_cases[1].translation="";
middle_cases[1].quantifier_functions="";
middle_cases[1].q_iterator_macros="";
   }
;
!literal
literal
: INTEGER_LITERAL
   {
!int
!n
literal.identifier=INTEGER_LITERAL.%text;
   }
| REAL_LITERAL
   {
!real
```

```
!n
literal.identifier=REAL_LITERAL.%text;
   }
| CHAR_LITERAL
   {
!char
!n
literal.identifier=CHAR_LITERAL.%text;
   }
| STRING_LITERAL
   {
!string
!n
literal.identifier=STRING_LITERAL.%text;
   }
| '#' NAME! enumeration type literal
   {
!enum
!n



   }
| '[' expressions ']'! sequence literal
   {
!sequence
!n



   }
| '{' expressions '}'! set literal
   {
!set
!n



   }
| '{' formals BIND expression '}'! set literal
   {
!set
!n



   }
| '{' expressions ';' expression '}'! map literal
   {
!map
!n
```

```
    }
| '[' pair_list ']'! tuple literal
    {
!tuple
!n



    }
| '{' NAME BIND expression '}'! union literal
    {
!union
!n



    }
;
    ! relation literals are sets of tuples
!expressions
expressions
: expression_list
    {



    }
|
    {



    }
;
!pair_list
pair_list
: pair_list ',' pair
    {



    }
| pair
    {



    }
;
```

```
!pair
pair
: name_list BIND expression
    {



    }
;
```

APPENDIX B

SPEC SUBSET IMPLEMENTED

This appendix contains the Spec Subset covered by the implementation. The complete Spec grammar was "pruned," resulting in the rules presented here. Complete production rules were removed, as well as some terminals and non-terminals. An asterisk ("*") in a production rule indicates that a non-terminal or terminal was pruned from the rule. Comments clarify the significance of the simplification. Assumptions and other information have been included as comments where it was considered prudent.

```
start
: spec
   {}
;
spec
: * module
   {!single module Specs only.}
;
module
| * * function * *
   {!functions only.}
;

function
: * FUNCTION module_header messages concepts END
   {!VIRTUAL Specs are not directly implemented and need not be considered.}
;
module_header
: formal_name
   {}
;

pragmas
            : pragmas PRAGMA actual_name '(' actuals ')'
               {PRAGMA update implemented to allow for procedure interface.}
```

```
;

messages
: * message
   {!Single service messages only.}
;

message
: MESSAGE formal_message pragmas response
   {}
;


response
: response_set
   {}
| response_cases
   {}
;

response_cases
: WHEN expression_list response_set response_cases
   {}
| OTHERWISE response_set
   {}
;

response_set
: * reply * *
   {!REPLY only, omits CHOOSE, SENDS, TRANSITION.}
;

reply
: REPLY actual_message where
   {!Non-generating REPLY only.}
;

formal_message
: * optional_formal_name formal_arguments
   {!No EXCEPTIONS in formal message (i.e., Functions with          no exception handling capability.).}
;

actual_message
: optional_exception optional_actual_name formal_arguments
   {!If an optional exception is used, no formal arguments are permitted.}
;

where
: WHERE expression_list
   {}
|
```

```
    {}
;

optional_exception
: EXCEPTION
    {}
|
    {}
;




concepts
: concepts concept
    {}
|
    {}
;

concept
        :CONCEPT formal_name '(' formals ')' VALUE '(' formals ')' where
    ! functions, defined with preconditions and postconditions
    {!Only concepts defined this way are currently permitted.}
;

optional_formal_name
: formal_name
    {}
|
    {}
;

formal_name
: NAME '{' formals '}'
    {}
| NAME
    {}
;

formal_arguments
: '(' formals ')'
    {}
|
    {}
;

formals
: field_list restriction
```

```
    {}
;

field_list
: field_list ',' type_binding
    {}
| type_binding
    {}
;

type_binding
: name_list ':' expression
    {!expression (i.e., the type) must be implemented in ADA. It may not be the spec generic
                type "t".}
;



name_list
: name_list NAME
    {}
| NAME
    {}
;

restriction
: SUCH expression_list
    {}
|
    {}
;

optional_actual_name
: actual_name
    {}
|
    {}
;

actual_name
: NAME '{' actuals '}'
    {}
            | NAME
    {}
;

actuals
: actuals ',' arg
    {}
| arg
```

```
    {}
;

arg
: expression
    {}
| pair
    {}
;

expression_list
: expression_list ',' expression
    {}
| expression
    {}
;

expression
        ! See Appendix G.
: actual_name ! variables and constants
    {}
| expression '(' actuals ')'! function call
    {}
| NOT expression
    {}
| expression AND expression
    {}
| expression OR expression
    {}
| expression '=' expression
    {}
| expression '<' expression
    {}
| expression '>' expression
    {}
| expression LE expression
    {}
| expression GE expression
    {}
| expression NE expression
    {}
| expression NLT expression
    {}
| expression NGT expression
    {}
| expression NLE expression
    {}
| expression NGE expression
    {}
| expression EQV expression
    {}
```

```
| expression NEQV expression
    { }
| '-' expression
    { }
| expression '+' expression
    { }
| expression '-' expression
    { }
| expression '*' expression
    { }
| expression '/' expression
    { }
| expression MOD expression
    { }
| expression EXP expression
    { }
| expression U expression
    { }
| expression APPEND expression
    { }
| expression IN expression
    { }
| '(' expression ')'
    { }
| literal ! literal with optional type id
    { }
;

literal
: INTEGER_LITERAL
    { }
| REAL_LITERAL
    { }
| CHAR_LITERAL
    { }
| STRING_LITERAL
    { }
;

expressions
: expression_list
    { }
|
    { }
;
```

APPENDIX C

USER'S MANUAL

I. INTRODUCTION

This system generates an Module Driver and Output Analyzer (MDOA) in Ada from a formal specification written in Spec. The system is extremely limited in its present form. It is capable of generating functioning MDOAs for a subset of the Spec function modules. That subset corresponds roughly to those modules that can be implemented as Ada functions or procedures and adhere to the specifications contained in this manual.

This manual provides the minimum details necessary to prepare for and execute a test using the Module Driver and Output Analyzer Generator. Several complete example sessions are included at the end of this manual.

II. USER REQUIREMENTS

The system depends upon the user providing or completing the following software components:

1. Module Specification (user provided).
2. Module Implementation (user provided).
3. Test Input Generator (user completed).
4. Test Iterators (user completed).
5. Test Criteria File (user provided).

Specifications and concrete interfaces for items 1 through 4 above are provided in Sections III through VII, respectively.

III. MODULE SPECIFICATION

The user must provide the module specification written in Spec. The file containing the specification must have a name ending with ".s" (e.g., spec_name.s). The MDOAG is very limited in its current form—only specifications which adhere to all of assertions below (should) yield a properly functioning MDOA.

- CONTAINS one (1) module.
- CONTAINS the keyword: FUNCTION.
- DOES NOT CONTAIN the keyword: VIRTUAL.
- CONTAINS only types in the Ada Standard Library or types declared in the implementation package.
- DOES NOT CONTAIN the keywords: IMPORT or INHERIT.
- CONTAINS one keyword: MESSAGE.
- The "formal message" DOES NOT CONTAIN the keyword: EXCEPTION.
- DOES NOT CONTAIN the keywords: CHOOSE, GENERATE, SEND, or TRANSITION.
- "EXCEPTION" responses DO NOT CONTAIN formal arguments.
- ALL CONCEPTS CONTAIN (1) a VALUE clause, "returning" a single formal of type boolean (2) a non-null WHERE clause starting with "b <=> ..." where b is the formal contained in the VALUE clause.
- DOES NOT CONTAIN expressions with the QUANTIFIERS: NUMBER, SUM, PRODUCT, SET, MAXIMUM, MINIMUM, UNION, INTERSECTION. (MAY CONTAIN: SOME and ALL.)
- DOES NOT CONTAIN previous state expressions (e.g., *x).

- DOES NOT CONTAIN collection of items expressions (e.g., $x).
- DOES NOT CONTAIN expressions of the form " '(' expression NAME ')' " (i.e., expressions with units of measurement).
- DOES NOT CONTAIN expressions with "DELAY."
- DOES NOT CONTAIN expressions with "PERIOD."
- DOES NOT CONTAIN expressions with "?."
- DOES NOT CONTAIN expressions with "!."
- DOES NOT CONTAIN literals with "#."
- DOES NOT CONTAIN **sequence** literals.
- DOES NOT CONTAIN **set** literals.
- DOES NOT CONTAIN **map** literals.
- DOES NOT CONTAIN **tuple** literals.
- DOES NOT CONTAIN **union** literals.

IV. MODULE IMPLEMENTATION

The user must provide the implementation of the Spec module in an Ada package. The implementation package must contain a subprogram that implements the specification and the I/O facilities required to read test criteria (e.g., values for generic parameters and unbounded test parameters) and write test results (input and output values).

A. Implementation Package

The module implementation must be contained in an Ada package in a file named "implementation.a." The implementation package name must be "IMPLEMENTATION." Figure C-1 shows a template for an implementation package. The statements enclosed in

asterisks represent code that must be provided by the implementor and are described below.

```
package IMPLEMENTATION is
        *Type Declarations*
        *Subprocedure Specification*
        *Exception Declarations*
        *I/O Subprocedures*
end IMPLEMENTATION;
```

Figure C-1. **Implementation Package Template**

1. **Type Declarations**

The implementor must declare all types used in the specification which are not contained in the Ada standard library. For example, if the Spec contains the type "real", then an Ada type "real" must be declared in the visible portion of the implementation package specification, or must be made visible via Ada "with" and "use" statements.

2. **Subprocedure Specification**

The subprocedure specification is the concrete interface to Spec module being implemented. It must be an Ada subprocedure specification whose name matches the Spec module name (module header NAME) (see Figure C-2). It must adhere to the concrete interface generation rules of Reference 1, pages 4-54 to 4-56. The parameters of the Ada formal part must correspond positionally to the formals

of the Spec formal message. When "update" pragmas are used, all formal message parameters are listed first, followed by reply parameters not included in an "update" pragma. If the Spec has generic parameters, then the Ada subprocedure specification must be generic. The Ada generic parameters must correspond positionally to the formals of the Spec module header. The Ada return "type mark" must correspond to the return type of the Spec REPLY parameter if the Spec calls for an Ada function.

```
--SAMPLE SPECIFICATION
FUNCTION foo(a b:type_1,c:type_2) --foo is generic, parms a,b,c
      MESSAGE(d  e:type_1)   --formals are d,e.
                                    .
                                    .
                                    .
      REPLY(f:another_type)

--CORRESPONDING SUBPROCEDURE SPECIFICATION
generic
      r: type_1;          --corresponds to a
      s: type_1;          --corresponds to b
      t: type_2;          --corresponds to c
function foo(x,y: type_1) return another_type;
    --x corresponds to d.
    --y corresponds to e.
    --Ada return type mark "another_type" corresponds to Spec
    --REPLY another_type.
```

Figure C-2. **Sample Spec Specification and Corresponding Ada Subprocedure Specification**

3. **Exception Declarations**

All exceptions in the message response of the Spec module must be declared in the visible portion of the implementation package specification. The Ada exception name (identifier) in the exception declaration must match the Spec exception name (actual name) (see Figure C-3).

```
--SPEC REPLY WITH EXCEPTION
REPLY EXCEPTION exception_foo

--CORRESPONDING Ada EXCEPTION DECLARATION
exception_foo: exception;        --same name as Spec EXCEPTION
```

Figure C-3. **Spec Reply with Exception and Corresponding Ada Declaration**

4. **I/O Subprocedures**

The specification of the implementation package must contain a "GET" operation for each distinct Spec generic parameter type. It must also contain a "PUT" statement for each distinct Spec MESSAGE/REPLY parameter type. The I/O Subprocedures corresponding to sample Spec of Figure C-2 are given in Figure C-4.

Notice that a "GET" is provided for each type of generic parameter (i.e., type_1, type_2) and that a "PUT" is provided for each type in the MESSAGE/REPLY parameters (i.e., type_1, another_type). A "PUT" is not required for type_2 because it is not a

```
procedure GET(outfile:  FILE_TYPE;  x:  type_1);
procedure GET(outfile:  FILE_TYPE;  x:  type_2);
procedure PUT(outfile:  FILE_TYPE;  x:  type_1);
procedure PUT(outfile:  FILE_TYPE;  x:  another_type);
```

Figure C-4. **I/O Subprocedures Corresponding to the Spec in Figure C-2**

MESSAGE/REPLY parameter. The "GET" and "PUT" must appear as shown in Figure C-4, except that additional parameters may follow the second parameter type mark. If additional parameters are added, associated default values must be provided for those parameters. The additional parameters are permitted to allow for definition of I/O in terms of the facilities provided in the Ada Standard Library. The defaults will always be used during test execution.

V. TEST INPUT GENERATOR

A test input generator shell is created by the MDOAG to match the Spec being tested. The user must complete it by providing the code necessary to generate the input values. The generator shell can be found in the file "input_generator.m4" after executing the "build_parts" command.

A shell generated for the Spec of Figure C-2 is shown in Figure C-5. The parameters "d" and "e" on line *2* are the variables for which values must be generated. The parameter "assurance" is the level of reliability desired of the module being tested—it is probability

```
include(/n/suns2/work/student/depasqua/MACROS/generator.m4)
**put with and use clauses here**--                              *1*
generator(GENERATOR, [assurance: float], [d:type_1;e:type_1],    *2*
[is
        --**put required declarations here**--                   *3*
begin
        --**put statements to generate values here**--           *4*
        generate(--**put generated values here**-);              *5*
        --**put more statements here as required**--             *6*
end GENERATOR;])
```

Figure C-5. **Sample "GENERATOR.M4" Shell Generated by MDOAG**

of error (i.e., [0.0, .1.0], assuming a random distribution of input values). The generator must be written so that a sufficiently large sequence of values is generated to achieve this level of reliability. (Note: The calculation of "n" in Figure C-6 provides a sufficiently large number meeting this criterion which may be used as a loop control bound.) Comments are included in the shell to assist the user. Detailed instructions follow:

- Replace line *1* with "with" and "use" statements as required to "import" packages/subprograms to support the generation of values (e.g., random number generator, math_pkg, etc.).
- Replace line *3* with Ada variable declaration statements required in the generation process.
- Replace line *4* with statements required to generate the values.
- Replace "--put...here--" on line *5* with the "generated" values. These become the actual parameters used in the test.
- Replace line *6* with any additional statements required for further generation (e.g., loop control, etc.).

```
include(/n/suns2/work/student/depasqua/MACROS/generator.m4)
with MATH; use MATH;
with TYPE_1_PKG; use TYPE_1_PKG;
generator(GENERATOR, [assurance: float], [d:type_1;e:type_1],
[is
 --n for loop control, ensures sufficient number of values are
 --generated to achieve "assurance" level testing
 n: constant natural :=
     natural((1.0/assurance)*float(BIN_LOG(1.0/assurance)));

 --r, s are the values to be generated corresponding to d and e
 --respectively.
 r, s: type_1;
begin
 for i in 1 .. n loop
     r := TYPE_1_PKG.RANDOM_NUMBER;
     s := TYPE_1_PKG.RANDOM_NUMBER;
     generate(r,s);
 end loop;
end GENERATOR;])
```

Figure C-6. **Sample Completed "GENERATOR.M4" Shell**

- Once completed, expand the macro into the file "generator.a" by executing the command "m4 generator.m4 > generator.a."
- Compile generator.a (ada generator.a).

Figure C-6 shows the same shell after completion by the user. In it the user conveniently takes advantage of a random number generator provided in "TYPE_1_PKG."

VI. TEST ITERATORS

One test ITERATOR shell is generated for each QUANTIFIER in the Spec. ITERATOR shells are identical in all physical regards to GENERATOR. They differ only in function. An ITERATOR must iterate

through the range of values of the parameters. It is left to the user to determine what sufficient coverage is. All ITERATOR shells of a Spec can be found in the "ITERATORS.m4" file. After the ITERATORS are completed and expanded (see GENERATOR), they should be compiled.

VII. TEST CRITERIA FILE

The user must provide the testing criteria. All tests require that the user provide an "assurance/reliability" level desired in the code. The "assurance" is a number between 0.0 and 1.0. The low end (0.0) implies high reliability (i.e., no more than 0 errors are produced each time the module is executed.). The high end (1.0) implies low reliability (i.e., no more than one error is produced each time the module is executed.). Values between 0.0 and 1.0 can be interpreted as "acceptable number of errors" per "number of executions" (e.g., 0.001 implies that 1 error every 1000 executions is acceptable).

In addition to the assurance, the user must provide other values required to perform testing. For example, the user must provide values to be used for generic instantiations. To determine the values required for the test, the user should examine the GET_TEST_PARAMETERS subprocedure of MAIN_PKG produced by the system. The sequence of "GET" statements indicates the values read by the system and required to conduct the test. Those values should be provided in the ASCII file "test_parameters," one value per line, in the order they appear in the sequence of read statements.

Multiple test cases can be run sequentially, without interruption, by providing a sequence of test input criteria in "test_parameters."

An example "MAIN_PKG.GET_TEST_PARAMETERS" subprocedure generated by the system and a "test_parameters" file that will cause two tests to be run are shown in Figure C-7. None of the comments will exist in the actual code nor are comments permitted in the "test_parameters" file. They are included in Figure C-7 for explanatory reasons only.

```
--SAMPLE GET_TEST_PARAMETERS OF PACKAGE MAIN_PKG
procedure GET_TEST_PARAMETERS is
begin
    FLT_IO.GET(INFILE, ASSURANCE);
    IMPLEMENTATION.GET(INFILE, PRECISION);
end GET_TEST_PARAMETERS;


--SAMPLE "test_parameters" FILE CAUSING TWO TEST EXECUTIONS
0.1      --Corresponds to "ASSURANCE" for first test
0.5      --Corresponds to "PRECISION" for first test
0.2      --Corresponds to "ASSURANCE" for second test
0.1      --Corresponds to "PRECISION" for second test
```

Figure C-7. **Sample GET_TEST_PARAMETERS Procedure and Corresponding "test_parameters" File**

VIII. THE ENVIRONMENT

The system makes several assumptions about its environment. Those assumptions are outlined in the following subparagraphs.

A. Ada Compiler

The system assumes that it exists in the user's Ada environment. The user's path should be set up to find the name "ada" to an Ada compiler.

B. Ada Library and its Contents

The system assumes that "IMPLEMENTATION", "GENERATOR" and any "ITERATORS" have been successfully compiled in the user's Ada library.

C. M4 Macro-processor and Macros

The system assumes the M4 macro processor is installed in the environment and that the macro "generator.m4" is accessible through in the path contained in the "include" statement at the top of the file "driver.m4" (produced by the system). If the path is not correct, the path in the program source code "check.k" should be changed to the proper path, and "check.k" should be recompiled. To recompile the source, execute the command "k check.k" from a directory in which the Kodiyak application generator is visible.

D. Command Files

The following commands should be visible in the user's environment: "check" (MDOAG object code), "build_parts" (script file invoking "check"), and "assemble_MDOA" (script file simulating "a.make"). This can be accomplished by adding /n/suns2/work/student/sepasqua/bin to your path variable (defined in your .cshrc file).

E. The Spec

The system assumes the Spec to be tested exists in the file "spec_name.s" in the current directory.

F. The Test Parameters

The system assumes the test parameters are in the text file "test_parameters" in the current directory.

IX. TEST RESULTS

Successful test runs generate a file "spec_name.err" containing test results similar to the example shown in Figure C-8. The results are broken into sections by test number. In the example, two tests were conducted. For test number 1, 500 sample inputs sets were tried and no errors were detected. For test number 2, 1000 sample inputs sets were tried and one error was found. It occurred when the input parameters had the values indicated in the figure and the return condition was normal (vice some exception condition). The postcondition (d + e = f) was not satisfied when the precondition (d = 0.00) was true, as specified (hypothetically), and was reported as an error. Summary statistics follow the final test. In the example, two tests were conducted, 1500 input data sets were run, resulting in the identification of one error.

X. USING THE SYSTEM

To generate and execute a MDOA, follow the steps below:

1. Ensure the environment conforms to Section VIII.
2. Ensure the Spec conforms to Section III.

```
**********************************************************
foo Test Results
**********************************************************
TEST NUMBER 1
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
INSTANCE SAMPLES TESTED:     500
INSTANCE ERRORS FOUND:         0
**********************************************************
TEST NUMBER 2
d = 0.000
e = 0.010
f = 0.221
condition = normal
WHEN d = 0.000 NOT (d + e = f)
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
INSTANCE SAMPLES TESTED:    1000
INSTANCE ERRORS FOUND:         1
**********************************************************
TOTAL TEST CONDUCTED:  2
TOTAL SAMPLES TESTED:  1500
TOTAL ERRORS FOUND:    1
**********************************************************
```

Figure C-8. **Sample Test Results**

3. Ensure the Implementation conforms to Section IV.

4. Execute the command: build_parts "spec_name.s" from the Unix prompt, where "spec_name.s" is the Spec to be tested.

5. Examine the subprogram "GET_TEST_PARAMETERS" of "MAIN_PKG" in the "MAIN_PKG.a" file. Provide "test_parameters" file conforming to Section VII.

6. Complete the generator shell: "GENERATOR.M4" as described in Section V.

7. Complete the iterator shells (if applicable): "ITERATORS.M4" as described in Section VI.

8. Execute the command "assemble_MDOA."

9. Execute the command "a.out" from the Unix prompt.

10. Examine the results ("spec_name.err") as described in Section IX.

SAMPLE

The following sample contains a Spec for a generic square root (Part A), the implementing package (Part B), the GET_TEST_PARAMETERS from MAIN_PKG.A (Part C), the test parameters supplied by the user (Part D), the user completed input generator (Part E), and the results of the test (Part F).

A. SPEC (SQUARE_ROOT.S)

```
FUNCTION square_root(precision:float SUCH THAT precision > 0.0)
        MESSAGE(x: float)
            WHEN x >= 0.0
                REPLY(y: float)
                WHERE y > 0.0, approximates(y * y, x)
            OTHERWISE REPLY EXCEPTION imaginary_square_root
        CONCEPT approximates(r1 r2: float)
            VALUE(b: boolean)
                WHERE b <=> abs(r1 - r2) <= abs(r2 * precision)
END
```

B. IMPLEMENTATION (IMPLEMENTATION.A)

```
with TEXT_IO; use TEXT_IO;
package IMPLEMENTATION is
   min_precision: constant float := float'epsilon;
   max_precision: constant float := float'last;
   subtype precision_type is float range min_precision .. max_precision;
   package FLT_IO is new FLOAT_IO(float);
   use FLT_IO;

   procedure PUT(outfile: FILE_TYPE; x:  float;
fore: field := DEFAULT_FORE;
aft: field := DEFAULT_AFT;
exp: field := DEFAULT_EXP) renames FLT_IO.PUT;
   procedure GET(infile: FILE_TYPE;  y:  out float;
width: FIELD := 0) renames FLT_IO.GET;

   generic
      precision: precision_type;
   function SQUARE_ROOT(x: float) return float;
   imaginary_square_root: exception;
```

```
end IMPLEMENTATION;

package body IMPLEMENTATION is
  function SQUARE_ROOT(x: float) return float is
    low, mid, midsq, high: float;
    tolerance: float := x * precision;
  begin
    if x < 0.0 then raise imaginary_square_root;
    elsif x = 0.0 then return x;
    end if;
    Henceforth x > 0.0.

    if x < 1.0 then low := x; high := 1.0;
    elsif x > 1.0 then low := 1.0; high := x;
    else return 1.0;
    end if;

    while (x - low * low) > tolerance loop
      Invariant: 0.0 < low ^ 2 < x < high ^ 2
      Bound: floor((high ^ 2 - low ^ 2) / tolerance
      mid := (high + low) * 0.5;
      midsq := mid * mid;

      if midsq > x then high := mid;
      elsif midsq < x then low := mid;
      else return mid;  -- mid ^ 2 = x
      end if;
    end loop;
    return low;
  end SQUARE_ROOT;
end IMPLEMENTATION;
```

C. GET_TEST_PARAMETERS FROM MAIN_PKG.A

```
procedure GET_TEST_PARAMETERS is
begin
FLT_IO.GET(INFILE,ASSURANCE);
IMPLEMENTATION.GET(INFILE,precision);
end GET_TEST_PARAMETERS;

procedure EXECUTE_TEST is
procedure NEW_DRIVER is new DRIVER(precision);
begin
NEW_DRIVER(ASSURANCE);
end EXECUTE_TEST;
```

D. TEST PARAMETERS FILE (TEST_PARAMETERS)

```
0.1       (The user decided to run two test sets.)
0.5
0.2
0.1
```

E. COMPLETED INPUT_GENERATOR.M4

```
include(/n/suns2/work/student/depasqua/MACROS/generator.m4)
**put with and use clauses here**--
with MATH; use MATH;
with RANDOM; use RANDOM;
generator(GENERATOR, [assurance:  float], [x:float],
[is
**put required declarations here**--
   n: constant natural :=
          natural((1.0/assurance)*float(BIN_LOG(1.0/assurance)));
   the_value:  float;
begin
**put statements to generate values here**--
   GET_TIME_SEED;
   for i in 1 .. n loop
the_value := RANDOM.NUMBER;
generate(the_value);
   end loop;
**put more statements here as required**--
end GENERATOR;])
```

F. TEST RESULTS (SQUARE_ROOT.ERR)

```
**********************************************************************
square_root Test Results
**********************************************************************


**********************************************************************
TEST NUMBER          1

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
INSTANCE SAMPLES TESTED:          33
INSTANCE ERRORS FOUND:             0
**********************************************************************
```

```
**********************************************************************
TEST NUMBER          2

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
INSTANCE SAMPLES TESTED:          12
INSTANCE ERRORS FOUND:             0
**********************************************************************

**********************************************************************
TOTAL TESTS CONDUCTED:             2
TOTAL SAMPLES TESTED:             45
TOTAL ERRORS FOUND:                0
**********************************************************************
```

APPENDIX D

SAMPLE SPEC, MDOA, IMPLEMENTATION, AND RESULTS

This Appendix contains a sample Spec, the MDOA generated by the MDOAG for that Spec, a faulty implementation which produces errors, the generator implementation, and the test results. A second example is located in the user's manual (Appendix C).

A. FUNCTION MAXIMUM

This example is a non-generic Spec for a function returning the maximum of two floats.

1. Spec (from user)

```
FUNCTION maximum
    MESSAGE (x y:float)
        WHEN x >= y
            REPLY (i:float) WHERE i = x
        OTHERWISE
            REPLY (i:float) WHERE i = y

END
```

2. MAIN PROGRAM (generated by build_parts or check command)

```
with REPORT;
with MAIN_PKG;
procedure MAIN is
begin
    REPORT.OPEN;
    while not (MAIN_PKG.TESTS_COMPLETE) loop
        MAIN_PKG.GET_TEST_PARAMETERS;
        MAIN_PKG.EXECUTE_TEST;
    end loop;
    REPORT.CLOSE;
end MAIN;
```

3. MAIN_PKG (generated by build_parts or check command)

```
package MAIN_PKG is
 function TESTS_COMPLETE return boolean;
 procedure GET_TEST_PARAMETERS;
 procedure EXECUTE_TEST;
end MAIN_PKG;

with FLT_IO;
with DRIVER;
with IMPLEMENTATION;
with TEXT_IO;
use TEXT_IO;
package body MAIN_PKG is
 INFILE    : FILE_TYPE;
 ASSURANCE : FLOAT range 0.0 .. 1.0;


 function TESTS_COMPLETE return boolean is
 begin
  if IS_OPEN(INFILE) and then END_OF_FILE(INFILE) then
   CLOSE(INFILE);
   return TRUE;
  elsif IS_OPEN(INFILE) then
   return FALSE;
  else
   OPEN(INFILE, IN_FILE, "test_parameters");
   return END_OF_FILE(INFILE);
  end if;
 end TESTS_COMPLETE;

 procedure GET_TEST_PARAMETERS is
 begin
  FLT_IO.GET(INFILE, ASSURANCE);
 end GET_TEST_PARAMETERS;

 procedure EXECUTE_TEST is
  procedure NEW_DRIVER(assurance : float) renames DRIVER;
 begin
  NEW_DRIVER(ASSURANCE);
 end EXECUTE_TEST;
end MAIN_PKG;
```

4. DRIVER.M4 (generated by build_parts or check command)

```
include(/n/suns2/work/student/depasqua/MACROS/generator.m4)
procedure DRIVER(assurance:  in float);
```

```
with GENERATOR;
with CHECK_PKG;
with REPORT;  use REPORT;
with IMPLEMENTATION;  use IMPLEMENTATION;
with CONDITION_TYPE_PKG; use CONDITION_TYPE_PKG;

procedure DRIVER(assurance: in float) is
condition:  condition_type := normal;
x, y: float; i: float;
function IMPLEMENTATION(x, y: float) return float renames maximum;
package BLACK_BOX is new CHECK_PKG(assurance);
begin
REPORT.WRITE_INSTANCE_HEADER;
foreach([x:float;y:float], GENERATOR,[assurance], [
begin
                                i := IMPLEMENTATION(x , y);
condition := normal;
exception

when others =>
              condition := unspecified_exception;
end;
BLACK_BOX.CHECK(condition, x, y, i);
INCREMENT_SAMPLES_TESTED;])
REPORT.WRITE_INSTANCE_STATS;
end DRIVER;
```

5. DRIVER.A (generated by m4)

```
with GENERATOR;
with CHECK_PKG;
with REPORT;  use REPORT;
with IMPLEMENTATION;  use IMPLEMENTATION;
with CONDITION_TYPE_PKG; use CONDITION_TYPE_PKG;

procedure DRIVER(assurance: in float) is
condition:  condition_type := normal;
x, y: float; i: float;
function IMPLEMENTATION(x, y: float)
                                        return float renames maximum;
package BLACK_BOX is new CHECK_PKG(assurance);
begin
REPORT.WRITE_INSTANCE_HEADER;
declare
  procedure loop_body(x:float;y:float) is
  begin
```

```
begin
                                    i := IMPLEMENTATION(x , y);
condition := normal;
exception

when others =>
     condition := unspecified_exception;
end;
BLACK_BOX.CHECK(condition,x, y, i);
INCREMENT_SAMPLES_TESTED;
  end loop_body;
  procedure execute_loop is new GENERATOR(loop_body);
begin
  execute_loop(assurance);
end;
REPORT.WRITE_INSTANCE_STATS;
end DRIVER;
```

6. CHECK_PKG (generated by build_parts or check command)

```
with REPORT;
use  REPORT;
with  IMPLEMENTATION;
use   IMPLEMENTATION;
with MDOAG_LIB;
use  MDOAG_LIB;
with CONDITION_TYPE_PKG;
use  CONDITION_TYPE_PKG;
generic
 assurance : float;
package CHECK_PKG is
 procedure  CHECK(condition : condition_type;
 x, y      : float;
 i         : float);
end CHECK_PKG;

package body CHECK_PKG is



 procedure CHECK(condition : condition_type;
 x, y      : float;
 i         : float) is
  preconditions_satisfied : boolean := false;
```

```
 begin
  if (x >= y) then
   if not ((i = x)) then
REPORT.ERROR(condition, x, y, i, "WHEN x>=y NOT i = x");
   end if;
   preconditions_satisfied := true;
  end if;
  if not (preconditions_satisfied) then
   if not ((i = y)) then
REPORT.ERROR(condition, x, y, i, "OTHERWISE  NOT i = y");
   end if;
  end if;
 end CHECK;
end CHECK_PKG;
```

7. REPORT_PKG (generated by check command)

```
with TEXT_IO;
use  TEXT_IO;
with  IMPLEMENTATION;
use   IMPLEMENTATION;
with CONDITION_TYPE_PKG;
use  CONDITION_TYPE_PKG;
package REPORT is
 procedure ERROR(condition : condition_type;
 x, y     : float;
 i        : float;
 msg      : string);
 procedure OPEN;
 procedure WRITE_INSTANCE_HEADER;
 procedure INCREMENT_SAMPLES_TESTED;
 procedure WRITE_INSTANCE_STATS;
 procedure CLOSE;
end REPORT;

package body REPORT is
 total_instances        : integer := 0;
 instance_samples_tested : integer := 0;
 total_samples_tested    : integer := 0;
 instance_errors         : integer := 0;
 total_errors           : integer := 0;
 outfile              : FILE_TYPE;
 package INT_IO is new INTEGER_IO(integer);
 use  INT_IO;
 package CONDITION_IO is new ENUMERATION_IO(CONDITION_TYPE);
 use  CONDITION_IO;
```

```
procedure OPEN is
begin
 CREATE(outfile, OUT_FILE, "maximum.err");
 for i in 1 .. 80 loop
  PUT(outfile, "*");
 end loop;
 NEW_LINE(outfile);
 PUT_LINE(outfile, "maximum Test Results");
 for i in 1 .. 80 loop
  PUT(outfile, "*");
 end loop;
 NEW_LINE(outfile);
 NEW_LINE(outfile);
end OPEN;
procedure WRITE_INSTANCE_HEADER is
begin
 total_instances := total_instances + 1;
 instance_errors := 0;
 instance_samples_tested := 0;
 NEW_LINE(outfile);
 for i in 1 .. 80 loop
  PUT(outfile, "*");
 end loop;
 NEW_LINE(outfile);
 PUT(outfile, "TEST NUMBER  ");
 PUT(outfile, total_instances);
 NEW_LINE(outfile);
end WRITE_INSTANCE_HEADER;

procedure INCREMENT_SAMPLES_TESTED is
begin
 instance_samples_tested := instance_samples_tested + 1;
 total_samples_tested := total_samples_tested + 1;
end INCREMENT_SAMPLES_TESTED;

procedure ERROR(condition : CONDITION_TYPE;
x, y     : float;
i        : float;
msg      : string) is
begin
 instance_errors := instance_errors + 1;
 total_errors := total_errors + 1;
 PUT(outfile, "ERROR:  ");
 NEW_LINE(outfile);
 PUT(outfile, msg);
 NEW_LINE(outfile);
 PUT(outfile, "x =");
```

```
  PUT(outfile, x);
  NEW_LINE(outfile);
  PUT(outfile, "y = ");
  IMPLEMENTATION.PUT(outfile, y);
  NEW_LINE(outfile);
  PUT(outfile, "i =");
  PUT(outfile, i);
  NEW_LINE(outfile);
  PUT(outfile, "Condition = ");
  PUT(outfile, condition);
  NEW_LINE(outfile);
end ERROR;
procedure WRITE_INSTANCE_STATS is
begin
  NEW_LINE(outfile);
  for i in 1 .. 40 loop
    PUT(outfile, "* ");
  end loop;
  NEW_LINE(outfile);
  PUT(outfile, "INSTANCE SAMPLES TESTED:   ");
  PUT(outfile, instance_samples_tested);
  NEW_LINE(outfile);
  PUT(outfile, "INSTANCE ERRORS FOUND:     ");
  PUT(outfile, instance_errors);
  NEW_LINE(outfile);
  for i in 1 .. 80 loop
    PUT(outfile, "*");
  end loop;
  NEW_LINE(outfile);
end WRITE_INSTANCE_STATS;

procedure CLOSE is
begin
  NEW_LINE(outfile);
  for i in 1 .. 80 loop
    PUT(outfile, "*");
  end loop;
  NEW_LINE(outfile);
  PUT(outfile, "TOTAL TESTS CONDUCTED:  ");
  PUT(outfile, total_instances);
  NEW_LINE(outfile);
  PUT(outfile, "TOTAL SAMPLES TESTED:    ");
  PUT(outfile, total_samples_tested);
  NEW_LINE(outfile);
  PUT(outfile, "TOTAL ERRORS FOUND:      ");
  PUT(outfile, total_errors);
  NEW_LINE(outfile);
```

```
  for i in 1 .. 80 loop
   PUT(outfile, "*");
  end loop;
  CLOSE(outfile);
 end CLOSE;
end REPORT;
```

8. CONDITION_TYPE_PACKAGE (generated by build_parts or check command)

```
with TEXT_IO;
use  TEXT_IO;
package CONDITION_TYPE_PKG is
 type CONDITION_TYPE is (normal, unspecified_exception);
 package CONDITION_TYPE_IO is new ENUMERATION_IO(CONDITION_TYPE);
end CONDITION_TYPE_PKG;
```

9. FAULTY IMPLEMENTATION (from user)

```
with TEXT_IO; use TEXT_IO;
package IMPLEMENTATION is
  package FLT_IO is new FLOAT_IO(float);
  use FLT_IO;

  procedure PUT(outfile: FILE_TYPE; x:  float;
fore: field := DEFAULT_FORE;
aft: field := DEFAULT_AFT;
exp: field := DEFAULT_EXP) renames FLT_IO.PUT;
  procedure GET(infile: FILE_TYPE;  y:  out float;
width: FIELD := 0) renames FLT_IO.GET;

  function maximum(x: float;y: float) return float;
end  IMPLEMENTATION;

package body IMPLEMENTATION is
  function maximum(x: float;y: float) return float is
   begin
if (x >= y) then
return y;   --FAULTY, SHOULD BE "return x"
else
return y;
end if;
     end maximum;
end  IMPLEMENTATION;
```

10. GENERATOR (interface generated by build_parts or check command and completed by the user)

```
include(/n/suns2/work/student/depasqua/MACROS/generator.m4)
**put with and use clauses here**--
with MATH; use MATH;
with RANDOM; use RANDOM;
generator(GENERATOR, [assurance:  float], [x:float;y:float],
[is
**put required declarations here**--
  n: constant natural := natural((1.0/assurance) *
                                 float(BIN_LOG(1.0/assurance)));
  x_gen, y_gen: float;
begin
**put statements to generate values here**--
      GET_TIME_SEED;
      for i in 1 .. n loop
          x_gen := RANDOM.NUMBER;
y_gen := RANDOM.NUMBER;
generate(x_gen, y_gen);
      end loop;
**put more statements here as required**--
end GENERATOR;])
```

11. INPUT FILE "test_parameters" (from user)

```
0.35
```

12. OUTPUT FILE "maximum.err"

```
**********************************************************************
maximum Test Results
**********************************************************************


**********************************************************************
TEST NUMBER          1
ERROR:
WHEN x>=y NOT i = x
x = 6.08648311665092E-01
y =  2.61542549659793E-01
i = 2.61542549659793E-01
Condition = NORMAL_CONDITION
ERROR:
WHEN x>=y NOT i = x
x = 7.27619776514044E-01
```

```
y =  6.95497007666803E-01
i = 6.95497007666803E-01
Condition = NORMAL_CONDITION

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
INSTANCE SAMPLES TESTED:      4
INSTANCE ERRORS FOUND:        2
**************************************************************************

**************************************************************************
TOTAL TESTS CONDUCTED:        1
TOTAL SAMPLES TESTED:         4
TOTAL ERRORS FOUND:           2
**************************************************************************
```

APPENDIX E

MACROS

This appendix contains the "foreach" and "generator" macros published as Reference 1, Appendix 7.

```
dnl -- m4 comments look like this
dnl -- use control characters for quotes to avoid conflicts with
dnl -- ada programs
changequote([,])
dnl -- input and output formats
dnl --
dnl -- format: generator(inputs, outputs, body)
dnl --
dnl -- sample invocation:
dnl --
dnl -- generator(g,[y1: t1; ... ; ym: tm],[x1: T1, ... , xn: Tn],
dnl -- [is
dnl --   -- declarations of g
dnl -- begin
dnl --   -- statements for calculating e1, ... , en
dnl --   generate(e1, ... , en);
dnl --   -- more statements, possibly containing "generate"
dnl -- end g;] )
dnl --
dnl -- expands to:
dnl --
dnl -- generic
dnl --   with procedure generate(x1: T1; ... ; xn: Tn);
dnl -- procedure g(y1: t1; ... ; ym: tm);
dnl -- procedure g(y1: t1; ... ; ym: tm) is
dnl --   -- declarations of g
dnl -- begin
dnl --   -- statements for calculating e1, ... , en
dnl --   generate(e1, ... , en);
dnl --   -- more statements, possibly containing "generate"
dnl -- end g;
dnl --
define([generator],
[generic
  with procedure generate($3);
```

```
procedure $1($2);

procedure $1($2) $4])
dnl --
dnl -- format: foreach(loop variables, generator name,
dnl --                 generator arguments,
dnl --                 statements in loop body)
dnl --
dnl -- sample invocation:
dnl --
dnl -- foreach([x1: T1, ... , xn: Tn], g, [e1, ... , em],
dnl --   [ -- sequence of statements in the loop body
dnl --   ] )
dnl --
dnl -- expands to:
dnl --
dnl -- declare
dnl --   procedure loop_body(x1: T1; ... ; xn: Tn) is
dnl --   begin
dnl --     -- sequence of statements in the loop body
dnl --   end loop_body;
dnl --   procedure execute_loop is new g(loop_body);
dnl -- begin
dnl --   execute_loop(e1, ... , em); -- translation of loop
dnl -- end;
dnl --
define([foreach],
[declare
  procedure loop_body($1) is
  begin
    $4
  end loop_body;
  procedure execute_loop is new $2(loop_body);
begin
  execute_loop($3);
end;])
dnl -- get rid of all predefined macro names
undefine([ifdef])
undefine([changequote])
undefine([divert])
undefine([undivert])
undefine([divnum])
undefine([ifelse])
undefine([incr])
undefine([eval])
undefine([len])
undefine([index])
```

```
undefine([substr])
undefine([translit])
undefine([include])
undefine([sinclude])
undefine([syscmd])
undefine([maketemp])
undefine([errprint])
undefine([dumpdef])
undefine([unix])
undefine([shift])
undefine([dnl])
undefine([define])
undefine([undefine])
```

APPENDIX F

TRANSLATION TEMPLATE SUMMARY

A. MAIN PACKAGE

```
package MAIN_PKG is
function TESTS_COMPLETE return boolean;
procedure GET_TEST_PARAMETERS;
procedure EXECUTE_TEST;
end MAIN_PKG;

with FLT_IO;
with DRIVER;
with IMPLEMENTATION;
with TEXT_IO; use TEXT_IO;
package body MAIN_PKG is
      INFILE:  FILE_TYPE;
      ASSURANCE:    FLOAT range 0.0..1.0;
      **GENERIC OBJECT DECLARATIONS**

      function TESTS_COMPLETE return boolean is
      begin

            if IS_OPEN(INFILE) and then END_OF_FILE(INFILE) then
         CLOSE(INFILE);
                  return TRUE;
  elsif IS_OPEN(INFILE) then
         return FALSE;
  else  OPEN(INFILE,IN_FILE,"test_parameters");
         return END_OF_FILE(INFILE);
  end if;
      end TESTS_COMPLETE;

      procedure GET_TEST_PARAMETERS is
      begin
            FLT_IO.GET(INFILE,ASSURANCE);
  **GENERIC OBJECT GETS**
      end GET_TEST_PARAMETERS;

      procedure EXECUTE_TEST is
            **DRIVER INSTANTIATION OR RENAMING DECLARATION**
      begin
  NEW_DRIVER(ASSURANCE);
      end EXECUTE_TEST;
end MAIN_PKG;
```

MAIN PACKAGE SUB-TEMPLATES

```
**GENERIC OBJECT DECLARATIONS**
Generic object declarations generated from Spec generic parameters
of the module header.  Objects may also be generated from messages
and concepts of the Spec where it is necessary to provide bounds
required provide bounds on the test.
e.g., precision:  float;


**GENERIC OBJECT GETS**
Facilitate input of the generic object.  Generates one per object.
IMPLEMENTATION.GET(INFILE,**GENERIC OBJECT #1**);
IMPLEMENTATION.GET(INFILE,**GENERIC OBJECT #2**);
        .
        .
        .
IMPLEMENTATION.GET(INFILE,**GENERIC OBJECT LAST**);


**DRIVER INSTANTIATION OR RENAMING DECLARATION**
**DRIVER INSTANTIATION**
Generated if generic objects are generated.
procedure NEW_DRIVER is new
                        DRIVER(**GENERIC OBJECT DECLARATIONS**);
OR
**DRIVER RENAMING DECLARATION**
Generated if no generic objects are generated.
  procedure NEW_DRIVER(assurance:  float) renames DRIVER;
```

B. DRIVER TEMPLATE

```
include(/n/suns2/work/student/depasqua/MACROS/generator.m4)
**GENERIC FORMAL PART**
procedure DRIVER(assurance:  in FLOAT);

with GENERATOR;
with CHECK_PKG;
with TYPES; use TYPES;
with REPORT; use REPORT;
with IMPLEMENTATION; use IMPLEMENTATION;
with CONDITION_TYPE_PKG; USE CONDITION_TYPE_PKG;

procedure DRIVER is (assurance: in float) is
condition:  condition_type := normal;
**PARAMETER SPECIFICATIONS**
**INSTANTIATIONS OR RENAMING DECLARATIONS**
begin
REPORT.WRITE_INSTANCE_HEADER;
```

```
foreach([("GENERATOR LOOP VARIABLES"), GENERATOR, [(assurance)], [
begin
      "FUNCTION CALL"
      condition := normal;
exception
      "EXCEPTION_WHEN_CLAUSES"
      when others =>
condition := unspecified_exception;
end;
BLACK_BOX.CHECK(condition,
"FORMAL MESSAGE ACTUAL PARMS");
INCREMENT_SAMPLES_TESTED;])
REPORT.WRITE_INSTANCE_STATS;
end DRIVER;
```

DRIVER SUB-TEMPLATES

"GENERIC FORMAL PART"
Generic formal part generated from the Spec generic parameters if they exist.
Otherwise no generic formal part is generated.

```
generic
   "GENERIC PARAMETER DECLARATIONS"
```

"PARAMETER SPECIFICATIONS"
Parameter specifications generated from the formal arguments of the Spec formal message "and" response.

"INSTANTIATIONS OR RENAMING DECLARATIONS"
Generated based on the type of ADA interface called for by the Spec and whether or not the function is generic.

non-generic function:

```
function IMPLEMENT("FORMAL MESSAGE PARM SPECIFICATIONS")
            return "TYPE MARK" renames "FUNCTION DESIGNATOR";
package BLACK_BOX is new CHECK_PKG(assurance);
```

non-generic procedure:

```
procedure IMPLEMENT("FUNCTION CALL SPECIFICATIONS")
             renames IMPLEMENTATION."FUNCTION DESIGNATOR";
package BLACK_BOX is new CHECK_PKG(assurance);
```

generic function:

```
function IMPLEMENT is new
            "FUNCTION DESIGNATOR"("GENERIC ACTUAL PARAMETERS");
package BLACK_BOX is new CHECK_PKG("GENERIC ACTUAL PARAMETERS",
      assurance);
```

generic procedure:

```
procedure IMPLEMENT is new
            "FUNCTION DESIGNATOR"("GENERIC ACTUAL PARAMETERS");
```

```
package BLACK_BOX is new CHECK_PKG(**GENERIC ACTUAL PARAMETERS**,
      assurance);
```

GENERATOR LOOP VARIABLES
Parameter Specifications generated from the Spec formal message, formated for the generator.m4 macro.

FUNCTION CALL
Function or procedure call depending on the ADA interface.

```
        function call:
        **REPLY CALL ACTUAL** :=
IMPLEMENT(**FORMAL MESSAGE CALL ACTUALS**);

procedure call:
**INITIALIZATION STATEMENTS**
IMPLEMENT(**CALL ACTUALS**);
```

FORMAL MESSAGE ACTUAL PARAMETERS
Actual parameters generated from the formal arguments of the Spec formal message.

EXCEPTION WHEN CLAUSES
When clauses generated from Spec response set exception clauses.

```
when **EXCEPTION ACTUAL NAME** =>
condition := **EXCEPTION_ACTUAL_NAME**_condition;
```

GENERIC PARAMETER DECLARATIONS
Generic parameter declarations generated from the Spec module header. May also be generated from other locations in the Spec in order to provide a means to bound otherwise unbounded variables.
e.g., precision: float;

FUNCTION DESIGNATOR
Function designator generated from the Spec module header NAME.

REPLY TYPE MARK
Type mark generated from the Spec reply type.

C. CHECK_PKG

```
include(/n/suns2/work/student/depasqua/MACROS/generator.m4)
with REPORT; use REPORT;
```

```
with MDOAG_LIB; use MDOAG_LIB;
with IMPLEMENTATION;  use IMPLEMENTATION;
with CONDITION_TYPE_PKG; use CONDITION_TYPE_PKG;
generic
      assurance:  float;
      **GENERIC OBJECT DECLARATIONS**
package CHECK_PKG is
      procedure CHECK(condition:  condition_type;
      **PARAMETER SPECIFICATIONS**);
end CHECK_PKG;

**QUANTIFIER WITH CLAUSES**
package body CHECK_PKG is

      **CONCEPT SUBPROGRAM SPECIFICATIONS**

      procedure CHECK(condition: condition_type;
      *PARAMETER SPECIFICATIONS**) is
            preconditions_satisfied: boolean := false;
            **QUANTIFIER FUNCTIONS**
      begin
  **RESPONSE TRANSFORMATION**
      end CHECK;

      **CONCEPT SUBPROGRAM BODIES**

end CHECK_PKG;

**GENERIC OBJECT DECLARATIONS**
See **GENERIC OBJECT DECLARATIONS** of MAIN PACKAGE

**PARAMETER SPECIFICATIONS**
See DRIVER TEMPLATE **PARAMETER SPECIFICATIONS**


**QUANTIFIER WITH CLAUSES**
With statements generated from Spec QUANTIFIER expressions.
Provide visibility to the ITERATOR required for quantifier evaluation. One "with" statement per
QUANTIFIER.
**QUANTIFIER TEXT** is actual quantifier NAME (e.g., ALL, SOME, etc.).
**QUANTIFIER LINE NUMBER** is the Spec line number the quantifier lies on.
The line number is used to ensure unique ITERATOR names.
with **QUANTIFIER TEXT**_**QUANTIFIER LINE NUMBER**;



**CONCEPT SUBPROGRAM SPECIFICATIONS**
Function specifications generated from Spec Concepts.
function **CONCEPT DESIGNATOR**(**CONCEPT PARM SPECS**) return **TYPE MARK**;
```

```
**QUANTIFIER FUNCTIONS**
Generated from Spec QUANTIFIER expressions.  One per QUANTIFIER.

**ALL QUANTIFIER FUNCTION**
Generated from Spec ALL QUANTIFIERS.
**QUANTIFIER PARAMETER SPECIFICATIONS**
function ALL_**QUANTIFIER LINE NUMBER** return boolean is
value:  boolean:= true;
**NESTED QUANTIFIER FUNCTIONS**
begin
foreach([**ITERATOR LOOP VARIABLES**],
  GEN_ALL_**QUANTIFIER LINE NUMBER**,
                                        [assurance],[
if (value = true) then
**QUANTIFIER ALL CHECKING TRANSLATION**
end if;])
return value;
end ALL_**QUANTIFIER LINE NUMBER**;


**SOME QUANTIFIER FUNCTION**
Generated from Spec SOME QUANTIFIERS.
**QUANTIFIER PARAMETER SPECIFICATIONS**
function SOME_**QUANTIFIER LINE NUMBER** return boolean is
value:  boolean:= false;
**NESTED QUANTIFIER FUNCTIONS**
begin
foreach([**ITERATOR LOOP VARIABLES**],
  GEN_SOME_**QUANTIFIER LINE NUMBER**,
                                        [assurance],[
if (value = false) then
**QUANTIFIER ALL CHECKING TRANSLATION**
end if;])
return value;
end SOME_**QUANTIFIER LINE NUMBER**;


**QUANTIFIER PARAMETER SPECIFICATIONS**
Parameter specifications generated from the declarations of the QUANTIFIER.


**ALL CHECKING TRANSLATION**
Generated from the ALL restrictions and expression list.
if **QUANTIFIER SUCH THAT TRANSLATION** then
if not **EXPRESSION TRANSLATION** then
value := false;
end if;
end if;

**SOME CHECKING TRANSLATION**
Generated from the SOME restrictions and expression list.
```

```
if **QUANTIFIER SUCH THAT TRANSLATION** then
if **EXPRESSION TRANSLATION** then
value := true;
end if;
end if;
```

****RESPONSE TRANSFORMATION** SUB-TEMPLATES**

****RESPONSE TRANSFORMATION****
Generated from Spec response.
****RESPONSE CASES TRANSFORMATION** OR **REPONSE SET TRANSFORMATION****

****RESPONSE CASES TRANSFORMATION****
Generated from Spec response cases.
****WHEN CASE TRANSFORMATION** OR **OTHERWISE CASE TRANSFORMATION****

```
**WHEN CASE TRANSFORMATION**
Generated when another WHEN condition follows this WHEN condition.
if **WHEN EXPRESSION LIST TRANSFORMATION** then
      **RESPONSE SET TRANSFORMATION**
      preconditions_satisfied := true;
end if;
```

```
**OTHERWISE CASE TRANSFORMATION**
Generated from Spec response case containing OTHERWISE clause.
This transformation is always an insert to the
**WHEN RESPONSE CASE TRANSFORMATION** shown above.
if not (preconditions_satisfied) then
      **RESPONSE SET TRANSFORMATION**
end if;
```

****RESPONSE SET TRANSFORMATION****
Generated from the reply of the Spec response set.
****EXCEPTION TRANSFORMATION** **WHERE EXPRESSION LIST TRANSFORMATION****

```
**EXCEPTION TRANSFORMATION**
Generated from the Spec reply EXCEPTION, if one exists.
if not (condition = **EXCEPTION NAME**_condition) then
REPORT.ERROR(condition, **RESPONSE ACTUAL PARAMETERS**,
      **WHEN ERROR MESSAGE**, "NOT EXCEPTION",
      **EXCEPTION NAME**);
end if;
```

```
**WHERE EXPRESSION LIST TRANSFORMATION**
Generated from the WHERE clause of the reply.
if not (**EXPRESSION #1 TRANSFORMATION**) then
                    REPORT.ERROR...
end if;
if not (**EXPRESSION #2 TRANSFORMATION**) then
                    REPORT.ERROR...
end if;
                              .
                              .
                              .

if not (**EXPRESSION #N  TRANSFORMATION**) then
                    REPORT.ERROR...
end if;



**WHEN EXPRESSION LIST TRANSFORMATION**
Generated from the WHEN clause of a Spec reply.
This is an insert to the **WHEN CASE TRANSFORMATION**
**EXPRESSION #1 TRANSFORMATION** and then
**EXPRESSION #2 TRANSFORMATION** and then
                              .
                              .

**EXPRESSION #N TRANSFORMATION**



**EXPRESSION TRANSFORMATION**
Appropriate ADA expression generated from a Spec expression.



**CONCEPT SUBPROGRAM BODIES** AND SUB-TEMPLATES
Generated from Spec concepts.


**CONCEPT SUBPROGRAM BODIES**
function **CONCEPT DESIGNATOR**(**CONCEPT PARM SPECS**) return **TYPE MARK**
                                                  is
**CONCEPT DECLARATIVE PART**
begin
**CONCEPT SEQUENCE OF STATEMENTS**
end **CONCEPT DESIGNATOR**


**CONCEPT PARM SPECS**
Generated from Spec Concept actual parameters.
```

CONCEPT DECLARATIVE PART
Generated from the Spec value type returned by the concept.

CONCEPT SEQUENCE OF STATEMENTS
Appropriate ADA statements generated from the WHERE clause of the concept.

CONCEPT SUBPROGRAM BODIES
function **CONCEPT DESIGNATOR**(**CONCEPT PARM SPECS**) return **TYPE MARK**
is
CONCEPT DECLARATIVE PART
begin
CONCEPT SEQUENCE OF STATEMENTS
end **CONCEPT DESIGNATOR**

SOME TEMPLATE
This template implements the SOME expression as a function so that it may be combined with the logic checking logic used in the system. It assumes a gemerator macro has been established which generates the values to be checked. It looks for a single case which proves the SOME to be true.

```
function some**S_NUMBER** return boolean is
      value: boolean := true
      **NESTED QUANTIFIER FUNCTIONS**
begin
      foreach([**GEN_LOOP_VARS**],GEN_SOME_**S_NUMBER**,[], [
            if (value = false) then
                  **SOME CHECKING CODE**
            end if;]);
            return value;
end some**S_NUMBER**;
```

S_NUMBER
A number concatenated to 'some' to eliminate naming conflicts.

NESTED QUANTIFIER FUNCTIONS
Functions corresponding to nested quantifiers of the SOME expression.

GEN_LOOP_VARS
See Driver Generator Loop Variables.

SOME CHECKING CODE
Analogous to **WHEN CASE TRANSFORMATION** of CHECK PACKAGE. It

checks to see if the value generated falls within the 'set' or 'range' restrictions. If so, it permits the values to checked. Otherwise, it short circuits the process. Note: this logic searches for case where the condition is true.

```
if **SUCH THAT TRANSFORMATION** then
        if **EXPRESSION TRANSFORMATION** then
                value = true;
        end if;
end if;
```

SUCH THAT TRANSFORMATION
Identical to **WHEN EXPRESSION LIST TRANSFORMATION** of CHECK PACKAGE.

ALL TEMPLATE
This template implements the ALL expression as a function so that it may be combined with the logic checking logic used in the system. It assumes a gemerator macro has been established which generates the values to be checked. It searches for one case which proves the ALL to be false.

```
function ALL**A_NUMBER** return boolean is
        value: boolean := true;
        **NESTED QUANTIFIER FUNCTIONS**
begin
        foreach([**GEN_LOOP_VARS**],GEN_ALL_**A_NUMBER**,[], [
                if (value = true) then
                        **ALL CHECKING CODE**
                end if;]);
                return value;
end all**A_NUMBER**;
```

A_NUMBER
A number concatenated to 'all' to eliminate naming conflicts.

NESTED QUANTIFER FUNCTIONS
Functions corresponding to nested quantifiers of the ALL expression.

GEN_LOOP_VARS
See Generator Loop Variables of DRIVER.

ALL CHECKING CODE
Analogous to **WHEN CASE TRANSFORMATION** of CHECK PACKAGE. It checks to see if the value generated falls within the 'set' or 'range' restrictions. If so, it permits the values to checked. Otherwise, it short circuits the process. Note: this logic searches for one false case to disprove the ALL.

```
if **SUCH THAT TRANSFORMATION** then
```

```
    if not **EXPRESSION TRANSFORMATION** then
        value = false;
    end if;
end if;
```

SUCH THAT TRANSFORMATION
Identical to **WHEN EXPRESSION LIST TRANSFORMATION** of CHECK PACKAGE.

D. REPORT PACKAGE TEMPLATE

```
with TEXT_IO; use TEXT_IO;
with IMPLEMENTATION; use IMPLEMENTATION;
with CONDITION_TYPE_PKG; use CONDITION_TYPE_PKG;
package REPORT is
procedure ERROR(condition: condition_type;
**PARAMETER SPECIFICATIONS**;
        msg: string);
procedure OPEN;
procedure WRITE_INSTANCE_HEADER(msg: string);
procedure INCREMENT_SAMPLES_TESTED;
procedure WRITE_INSTANCE_STATS;
procedure CLOSE;
end REPORT;

package body REPORT is
        instance_samples_tested: integer := 0;
        total_samples_tested: integer := 0;
instance_errors: integer := 0;
total_errors: integer := 0;
outfile: FILE_TYPE;
package INT_IO is new INTEGER_IO(integer);
use INT_IO;
package CONDITION_IO is
                new ENUMERATION_IO(CONDITION_TYPE);
use CONDITION_IO;

procedure OPEN is
begin
                    CREATE(outfile, OUT_FILE, "**FUNCTION NAME**.err");
for i in 1..80 loop PUT(outfile,"*"); end loop;
NEW_LINE(outfile);
PUT_LINE(outfile,
                        "**FUNCTION DESIGNATOR** Test Results");
for i in 1..80 loop PUT(outfile,"*"); end loop;
NEW_LINE(outfile);
NEW_LINE(outfile);
```

```
end OPEN;

procedure WRITE_INSTANCE_HEADER(msg: in string) is
begin
instance_errors := 0;
instance_samples_tested := 0;
NEW_LINE(outfile);
for i in 1..80 loop PUT(outfile,"*"); end loop;
NEW_LINE(outfile);
PUT(outfile,msg);
NEW_LINE(outfile);
for i in 1..40 loop PUT(outfile,"* "); end loop;
NEW_LINE(outfile);
end WRITE_INSTANCE_HEADER;

procedure INCREMENT_SAMPLES_TESTED is
begin
instance_samples_tested :=
                              instance_samples_tested + 1;
total_samples_tested :=
                              total_samples_tested + 1;
end INCREMENT_SAMPLES_TESTED;

procedure ERROR(condition: CONDITION_TYPE;
**PARAMETER SPECIFICATIONS**;
msg: string) is
begin
instance_errors := instance_errors + 1;
total_errors := total_errors + 1;
PUT(outfile,"ERROR: ");
NEW_LINE(outfile);
  PUT(outfile,msg);
NEW_LINE(outfile);
**PARAMETER PUT STATEMENTS*
PUT(outfile,"Condition = ");
PUT(outfile,condition);
NEW_LINE(outfile);
end ERROR;

procedure WRITE_INSTANCE_STATS is
begin
NEW_LINE;
for i in 1..40 loop PUT(outfile,"* "); end loop;
NEW_LINE(outfile);
PUT(outfile,"INSTANCE SAMPLES TESTED: ");
PUT(outfile,instance_samples_tested);
NEW_LINE(outfile);
PUT(outfile,"INSTANCE ERRORS FOUND: ");
PUT(outfile,instance_errors);
NEW_LINE(outfile);
for i in 1..80 loop PUT(outfile,"*"); end loop;
```

```
NEW_LINE(outfile);
end WRITE_INSTANCE_STATS;

procedure CLOSE is
begin
NEW_LINE(outfile);
for i in 1..80 loop PUT(outfile,"="); end loop;
NEW_LINE(outfile);
PUT(outfile,"TOTAL SAMPLES TESTED: ");
PUT(outfile,total_samples_tested);
NEW_LINE(outfile);
PUT(outfile,"TOTAL ERRORS FOUND: ");
PUT(outfile,total_errors);
NEW_LINE(outfile);
for i in 1..80 loop PUT(outfile,"="); end loop;
CLOSE(outfile);
end CLOSE;
end REPORT;
```

REPORT PACKAGE SUB-TEMPLATES

"PARAMETER SPECIFICATIONS"
Generated from the actuals of the Spec formal message.

"FUNCTION DESIGNATOR"
Generated from the Spec function NAME of the module header.

"PARAMETER PUT STATEMENTS"
Generated from Spec actual message parameters.

```
PUT(outfile,"**MESSAGE PARAMETER #1** = ");
PUT(outfile,**MESSAGE PARAMETER #1**);
NEW_LINE(outfile);
PUT(outfile,"**MESSAGE PARAMETER #2** = ");
PUT(outfile,**MESSAGE PARAMETER #2**);
NEW_LINE(outfile);
.
.
.
PUT(outfile,"**MESSAGE PARAMETER #N** = ");
PUT(outfile,**MESSAGE PARAMETER #N**);
NEW_LINE(outfile);
```

E. CONDITION_TYPE_PKG

```
with TEXT_IO; use TEXT_IO;
package CONDITION_TYPE_PKG is
        type CONDITION_TYPE is (**CONDITION TYPES**);
        package        CONDITION_TYPE_IO        is        new
                          ENUMERATION_IO(CONDITION_TYPE);
end CONDITION_TYPE_PKG;
```

F. GENERATOR

```
include(/n/suns2/work/student/depasqua/MACROS/generator.m4)
**put with and use clauses here**--
generator(**GEN_FUNCTION_DESIGNATOR**, [assurance: float],
                [**GENERATOR LOOP VARIABLES**],
 [is
                **put required declarations here**--
 begin
                **put statements to generate values here**--
                generate(--**put generated values here**-);
                **put more statements here as required**--
 end **GEN_FUNCTION_DESIGNATOR**;])
```

G. ITERATOR

```
include(/n/suns2/work/student/depasqua/MACROS/generator.m4)
**put with and use statements here**--
generator(GEN_SOME_5,[assurance: float],[r:type_2],
[is
**put any required declarations here**--
begin
**put iterating statements here**--
generate(--**put generated values here**);
**put more statements here as required**--
end GEN_SOME_5;])

**put with and use statements here**--
generator(GEN_ALL_4,[assurance: float],[q:type_1],
[is
**put any required declarations here**--
begin
**put iterating statements here**--
generate(--**put generated values here**);
**put more statements here as required**--
end GEN_ALL_4;])
```

APPENDIX G

EXPRESSION TRANSLATION SUMMARY

This appendix contains a summary of the Spec-to-Ada expression translations used in the MDOAG. Assumptions were made in choosing the translations. A careful examination of these translations is helpful for MDOAG users. (Note: "e" and "et" are shorthand for "expression" and "expression translation," respectively. In the examples, x => y denotes "x" translated to "y." Where Spec expressions consist of multiple expressions, positional correspondence of the expressions to translations holds (e.g., The first and second "e" correspond to the first and second "et," respectively, in: e op e => et op et.).

SPEC EXPRESSION	TRANSLATION
1. QUANTIFIER (formals :: e) --(e.g., ALL(x:nat SUCH THAT p(x) :: q(x)) => ALL_XX WHERE --ALL_XX is a function call returning a boolean indicating --the satification or dissatifaction of the QUANTIFER. XX --is the Spec line number the QUANTIFER exists on.)	See CHECK_PKG Template.
2. actual_name --variables, constants (e.g., 5.0 => 5.0)	actual_name.text
3. e (actuals)	et (actual.actual_parms) --function call (e.g., max(x+1,5) => max((x+1),(5))
4. e @ actual_name --assumes overloaded enumeration type for which Ada resolves --as translated [Ref. 6:p. 381]. --(e.g., red@color => color'(red)) --Spec explicit type cast for overloaded enumeration types --(e.g., red@color, red@traffic_light) or literals of --relation types (e.g., {[a::1,b::2],[a::2,b::4]}@set(tuple(a	actual_name.text'(et)

--b:integer}})

5. ~ e — NOT (et)

6. e & e — (et AND et)

7. e | e — (et OR et)

8. e => e — implies(et,et)
--implemented as a function in MDOAG_library.

9. e <=> e iff(et,et)
--implemented as a function in MDOAG_library.

10. e = e — (et = et)

11. e < e — (et < et)

12. e > e — (et > et)

13. e <= e — (et <= et)

14. e >= e — (et >= et)

15. e ~= e — (et \= et)

16. e ~< e — NOT (et < et)

17. e ~> e — NOT (et > et)

18. e ~<= e — NOT (et <= et)

19. e ~>= e — NOT (et >= et)

20. e == e — equivalent(et, et)
--assumes user defined in IMPLEMENTATON.

21. e ~== e — NOT equivalent(et,et)
--See comment for translation 20.

22. - e — - et

23. e + e — (et + et)

24. e - e — (et - et)

25. e * e — (et * et)

26. e / e (et / et)

27. e MOD e (et MOD et)

28. e ^ e (et ** et)
--assumes ^ implemented as **.

29. e U e union(et, et)
--assumes U defined for the type of et.

30. e || e (et & et)
--assumes Spec concat implemented as Ada concat.

31. e IN e (et IN et)

32. * expr --not implemented.

33. $ expr --not implemented.

34. e .. e (et .. et)

35. e.NAME et.NAME
--assumes implementation by Ada record.

36. e[e] et(et)
--assumes implemetation by Ada array.

37. (e) (et)

38. (e NAME) (et NAME)
--checking routines are not implemented for this.

39. TIME CLOCK
--assumes system clock = local time desired.

40. DELAY --not implemented.
--(30 DAYS <= DELAY <= 31 DAYS) [Ref. 1:p. 3-116].

41. PERIOD --not implemented.
--PERIOD = (14 DAYS) [Ref. 1:p. 3-117].

42. literal --As shown below.
--REAL, INTEGER, STRING, CHAR literal.text
--# NAME NAME
--all others --not implemented.

43. ? --not implemented.

```
44.  !                                          --not implemented.

45.  IF e1 THEN e2 middle_cases ELSE e3 FI
                                                --See CHECK_PKG template.
     --assumes "stand-alone" use in Spec responses of the form:
     --WHERE IF e1 THEN e2
     --      ELSE_IF e3 THEN e4
     --      ELSE e5 FI
     --e_list -> e_list, e
     --      where e is      --if-then-else-fi expression is not
     --supported.
     --e1 -> e2 op e3
     --      e2 & e3 may not be if-then-else-fi expression.
     --General Form of the translation:
     --if et1 then
     --      if not et2 then "error" end if;
     --middle_cases_trans (elsif's using same logic)
     --else
     --      if not et3 then "error" end if;
     --end if;
```

LIST OF REFERENCES

1. Berzins, V., and Luqi, *Draft of Software Engineering with Abstractions: An Integrated Approach to Software Development with ADA*, Addison-Wesley, 1990.

2. ANSI/MIL-STD-1815A-1983, *Reference Manual for the ADA Programming Language*, United States Department of Defense, ADA Joint Program Office, February 1983.

3. Beizer, B., *Software Testing Techniques*, Van Nostrand Reinhold Company, Inc., 1983.

4. Liebein, E., *The Department of Defense Software Initiative- A Status Report*, Communications of the ACM, v. 29, n. 8, p. 734, August 1986.

5. Boehm, B. W. and Papaccio, P. N., *Understanding and Controlling Software Costs*, IEEE Transactions of Software Engineering, v. 14, n. 10, pp. 1462-1470, October 1988.

6. Booch, G. B., *Software Engineering with ADA*, 2nd ed., pp. 420-421, Benjamin/Cummings Publishing Company, Inc., 1987.

7. Lamb, D. A., *Software Engineering: Planning for Change*, Prentice Hall, 1988.

8. Naval Postgraduate School Technical Report, NPS52-89-029, *A Student's Guide to Spec*, by V. Berzins and R. Kopas, May 1989.

9. Berzins, V. and Luqi, *An Introduction to the Specification Language SPEC*, IEEE Transactions on Software, v. 7, no. 2, March 1990.

10. Knuth, D. E., *Semantics of Context-Free Languages*, Mathematical Systems Theory, v. 2, n. 2, pp. 127-133, November 1967.

11. University of Minnesota Computer Science Technical Report 85-37, *The Incomplete AG User's Guide and Reference Manual*, by R. M. Herndon, Jr., October 1985.

12. Bell Laboratories Computer Science Technical Report 39, *Lex A Lexical Analyzer Generator*, by M. Lesk and E. Schmidt, October 1975.

13. Myers, G. J., *The Art of Software Testing*, pp. 148-149, John Wiley & Sons, Inc., 1979.

14. Panzel, D. J., *Automatic Software Test Drivers*, Computer, v. 11, n. 4, pp. 44-50, April 1978.

15. Altizer, C., *Implementation of a Language Translator for the Computer Aided Prototyping System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

16. Kopas, R. G., *The Design and Implementation of a Specification Language Type Checker*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1989.

17. Luckham, D. C., and others, *ANNA A Language for Annotating Ada Programs*, Springer-Verlag, 1987.

BIBLIOGRAPHY

Aho, V. A., Sethi, R., and Ullman, J. D., *Compilers Principles, Techniques, and Tools*, Addison Wesley, 1988.

Brookshear, J. G., *Theory of Computation Formal Languages, Automata, and Complexity*, Benjamin Cummings, 1989.

Farrow, R., "Generating a Production Compiler for an Attribute Grammar," *IEEE Software*, v. 1, pp. 77-93, October 1984.

Howden, W. E., *Functional Program Testing & Analysis*, McGraw-Hill, 1987.

Skansholm, J., *Ada from the Beginning*, Addison Wesley, 1988.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Director of Research Administration Code 012 Naval Postgraduate School Monterey, CA 93943	1
4.	Chairman, Code 52 Computer Science Department Naval Postgraduate School Monterey, CA 93943	1
5.	Office of Naval Research 800 N. Quincy Street Arlington, VA 22217-5000	1
6.	Center for Naval Analysis 4401 Ford Avenue Alexandria, VA 22302-0268	1
7.	National Science Foundation Division of Computer and Computation Research Washington, D.C. 20550	1
8.	Commandant of the Marine Corps Code TE06 Headquarters, U.S. Marine Corps Washington, D.C. 20380-0001	1
9.	Office of the Chief of Naval Operations Code OP-941 Washington, D.C. 20350	1

10. Office of the Chief of Naval Operations — 1
Code OP-945
Washington, D.C. 20350

11. Commander Naval Telecommunications Command — 2
Naval Telecommunications Command Headquarters
4401 Massachusetts Avenue NW
Washington, D.C. 20390-5290

12. Commander Naval Data Automation Command — 1
Washington Navy Yard
Washington, D.C. 20374-1662

13. Dr. Lui Sha — 1
Carnegie Mellon University
Software Engineering Institute
Department of Computer Science
Pittsburgh, PA 15260

14. COL C. Cox, USAF — 1
JCS (J-8)
Nuclear Force Analysis Division
Pentagon
Washington, D.C. 20318-8000

15. Commanding Officer — 1
Code 5150
Naval Research Laboratory
Washington, D.C. 20375-5000

16. Defense Advanced Research Projects Agency (DARPA) — 1
Integrated Strategic Technology Office (ISTO)
1400 Wilson Boulevard
Arlington, VA 22209-2308

17. Defense Advanced Research Projects Agency (DARPA) — 1
Director, Naval Technology Office
1400 Wilson Boulevard
Arlington, VA 2209-2308

18. Defense Advanced Research Projects Agency (DARPA) — 1
Director, Prototype Projects Office
1400 Wilson Boulevard
Arlington, VA 2209-2308

19. Defense Advanced Research Projects Agency (DARPA) 1
Director, Tactical Technology Office
1400 Wilson Boulevard
Arlington, VA 2209-2308

20. Dr. R. M. Carroll (OP-01B2) 1
Chief of Naval Operations
Washington, DC 20350

21. Dr. Aimram Yehudai 1
Tel Aviv University
School of Mathematical Sciences
Department of Computer Science
Tel Aviv, Israel 69978

22. Dr. Robert M. Balzer 1
USC-Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, CA 90292-6695

23. Dr. Ted Lewis 1
Editor-in-Chief, IEEE Software
Oregon State University
Computer Science Department
Corvallis, OR 97331

24. Dr. R. T. Yeh 1
International Software Systems Inc.
12710 Research Boulevard, Suite 301
Austin, TX 78759

25. Dr. C. Green 1
Kestrel Institute
1801 Page Mill Road
Palo Alto, CA 94304

26. Prof. D. Berry 1
Department of Computer Science
University of California
Los Angeles, CA 90024

27. Director, Naval Telecommunications 1
System Integration Center
NAVCOMMUNIT Washington
Washington, D.C. 20363-5110

28. Dr. Knudsen 1
Code PD50
Space and Naval Warfare Systems Command
Washington, D.C. 20363-5110

29. Ada Joint Program Office 1
OUSDRE(R&AT)
The Pentagon
Washington, D.C. 23030

30. CAPT A. Thompson 1
Naval Sea Systems Command
National Center #2, Suite 7N06
Washington, D.C. 22202

31. Dr. Peter Ng 1
New Jersey Institute of Technology
Computer Science Department
Newark, NJ 07102

32. Dr. Van Tilborg 1
Office of Naval Research
Computer Science Division, Code 1133
800 N. Quincy Street
Arlington, VA 22217-5000

33. Dr. R. Wachter 1
Office of Naval Research
Computer Science Division, Code 1133
800 N. Quincy Street
Arlington, VA 22217-5000

34. Dr. J. Smith, Code 1211 1
Office of Naval Research
Applied Mathematics and Computer Science
800 N. Quincy Street
Arlington, VA 22217-5000

35. Dr. R. Kieburtz 1
Oregon Graduate Center
Portland (Beaverton)
Portland, OR 97005

36. Dr. M. Ketabchi 1
Santa Clara University
Department of Electrical Engineering and Computer Science
Santa Clara, CA 95053

37. Dr. L. Belady 1
Software Group, MCC
9430 Research Boulevard
Austin, TX 78759

38. Dr. Murat Tanik 1
Southern Methodist University
Computer Science and Engineering Department
Dallas, TX 75275

39. Dr. Ming Liu 1
The Ohio State University
Department of Computer and Information Science
2036 Neil Ave Mall
Columbus, OH 43210-1277

40. Mr. William E. Rzepka 1
U.S. Air Force Systems Command
Rome Air Development Center
RADC/COE
Griffis Air Force Base, NY 13441-5700

41. Dr. C.V. Ramamoorthy 1
University of California at Berkeley
Department of Electrical Engineering and Computer Science
Computer Science Division
Berkeley, CA 90024

42. Dr. Nancy Levenson 1
University of California at Irvine
Department of Computer and Information Science
Irvine, CA 92717

43. Dr. Mike Reiley 1
Fleet Combat Directional Systems Support Activity
San Diego, CA 92147-5081

44. Dr. William Howden 1
University of California at San Diego
Department of Computer Science
La Jolla, CA 92093

45. Dr. Earl Chavis (OP-162) 1
Chief of Naval Operations
Washington, DC 20350

46. Dr. Jane W. S. Liu 1
University of Illinois
Department of Computer Science
Urbana Champaign, IL 61801

47. Dr. Alan Hevner 1
University of Maryland
College of Business Management
Tydings Hall, Room 0137
College Park, MD 20742

48. Dr. Y. H. Chu 1
University of Maryland
Computer Science Department
College Park, MD 20742

49. Dr. N. Roussapoulos 1
University of Maryland
Computer Science Department
College Park, MD 20742

50. Dr. Alfs Berztiss 1
University of Pittsburgh
Department of Computer Science
Pittsburgh, PA 15260

51. Dr. Al Mok 1
University of Texas at Austin
Computer Science Department
Austin, TX 78712

52. George Sumiall 1
US Army Headquarters
CECOM
AMSEL-RD-SE-AST-SE
Fort Monmouth, NJ 07703-5000

53. Mr. Joel Trimble 1
1211 South Fern Street, C107
Arlington, VA 22202

54. Linwood Sutton 1
Code 423
Naval Ocean Systems Center
San Diego, CA 92152-5000

55. Dr. Sherman Gee — 1
Code 221
Office of Naval Technology
200 N. Quincy Street
Arlington, VA 22217

56. Dr. Mario Barbacci — 1
Carnegie-Mellon University
Software Engineering Institute
Pittsburgh, PA 15213

57. Dr. Mark Kellner — 1
Carnegie-Mellon University
Software Engineering Institute
Pittsburgh, PA 15213

58. Dr. Luqi — 10
Code 52Lq
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

59. Dr. V. Berzins — 2
Code 52Bz
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

60. Captain G. A. DePasquale — 2
12 Vista Woods Road
Stafford, VA 22554